PLOTTER

Ibrahima Niang	in2190
Ranjith Kumar Shanmuga Visvanathan	rs3579
Sania Arif	sa3311

10th May, 2016.

Contents

Part I
Introduction

plOtter is a data manipulation language built on the principles and appreciation of a minimalist aesthetic. The goal of plOtter is to provide a means for users to design and implement their own plots to visualize data, that would otherwise be done by rigid, automated software (such as Microsoft Excel). Our language will simplify charting maps through domain-specific types and operations. The users can build their custom graph templates using our language and can reuse the template to visualize different data points.
Very primitive examples could be building a bar-graph, histogram etc. Higher level examples could be drawing small album art, or other complex examples could be to visualize/plot a binary tree, Gantt chart, etc.
Following is a manual that provides reference information for using plOtter. It describes the lexical conventions, basic types scoping rules, built-in functions and also displays a sample program and output.

Part II
Tutorial

This covers you through the basic things in starting with Plotter.

1 Environment Setup

The environment required are Linux based, Ocaml and g++ 11 compiler. Install OCaml and g++ 11, as the target language for Plotter is c++ and its compiled with g++ 11.
Then the Plotter source code can be cloned from
¿ git clone https://github.com/saniaarif22/PlOtter.git

2 Using the Compiler

The code is inside src/ folder. Follow the README.md for information and crunched steps.
Run 'make' inside src, to build the Plotter.
Then run your .plt code with

And Voila! This will generate the svg at the same path of your .plt file, if it compiled successfully and throws the corresponding errors if it didnt.

3 Basics

This sections guides you through some of the basics of Plotter and implementing them in Plotter.

3.1 Primitives

The primitives supported are num, string, bool and point. An example program showing declaration and usage of primitive variables is shown.

```
num a
bool b
string c
point d
a = 10
b = true
c = "Hello Word!"
d = (a,20)
```

Listing 1: "Primitives in Plotter"

3.2 List

We apart from primitive also support non primitive data structure. The main this is List. List can contain any of the primitive as elements. A small sample is given below to

show the declaration and usage of list with its operations.

```
#List of num elements
list num a

#setting it variables
a = [10,3,8,4,10]

#Append element to it
a.append(20)

#pop an element
a.pop()  /* a is now [10,3,8,4,10] */

#remove
a.remove(1) /* a is now [10,8,4,10] .. a[1] is removed*/

#similar operations with bool or string
list bool b
list string c

#setting it variables
b = [10,3,8,4,10]
c = [ "this", "is", "a", "sample"]
```

Listing 2: "Lists in Plotter"

3.3 Operators

Operators are listed in order of precedence. All operators associate left to right, except for assignment, which associates right to left.

Arithmetic	+, -, *, /, %, **
Comparison	==, ! =, <, >, <=, >=, and, or
Assignment	=, + =, − =, ∗ =, / =, % =, ∗ =
Membership	in, not in

```
#List of num elements
num a
a = 10
a = a+1 *(5-4)
if a > 0 && a <10 :
    print "Between 0 and 10"
else:
    print "Not between 0 and 10"
end
```

Listing 3: "Operators in Plotter"

3.4 Loops

Plotter provides both for loop and while loop. A sample for and while loop in Plotter.

3.5 Functions

In Plotter, functions are supported which comes in handy when we want to manipulate data or creating your own plotting functions like draw rectangle, draw square etc. Also the internal library contains full of functions like these.

```

fn printAll(list num a):
    num i
    for i=0;i<a.length();i++:
        print i
    end
end

#List of num elements
list num a
a = [3,4,5,6]
printAll(a)
```

Listing 4: "Functions in Plotter"

3.6 Line

This is the essence of the language. The ability to draw beautiful images, and in anyway we want. The very core available in-built function to do this is Line.

```

fn rect(point a, num h, num w):
    line(a,(a[0]+w,a[1]))
    line(a,(a[0],a[1]+h))
    line((a[0]+w,a[1]),(a[0]+w,a[1]+h))
    line((a[0],a[1]+h),(a[0]+w,a[1]+h))
end
```

Listing 5: "Using built-in line"

Below shown is a simple implementation of drawing a rectangle with line.
We also support drawing the same line with color. Below shown is a implementation of filled rectagle with colored lines.

```
fn rectFill(point a, num h, num w, string color):
    num i
    point x
    point y
    string s

    /* Make a rectanle by drawing multiple lines */
    for i=0;i<w;i=i+1:
        x = (a[0]+i, a[1])
        y = (a[0]+i, a[1]+h)
        line( x , y , color )
    end
end
```

Listing 6: "Using built-in line with color"

3.7 Typography

The second important feature when is comes to plotting is printing the text. By default we support two inbuilt functions. They are print and printXY.

3.7.1 print

The print, can be used to print and debug simple things, it posts the print expr in the middle of the svg. It might not be very advantageous in printing lots of data, as they may over lap.

```
num a
a = 10
print 4
print a
print "Hello"
```

Listing 7: "Simple Print"

3.7.2 printXY

The printXY, can be used to print an expression at a specific (x,y) coordinate in the output SVG. This comes really helpful in printing things like axes, lables, titles etc.

```
num a
a = 10
#Prints 4 at location (10,10)
print (4, (10,10))
point p
p = (100.5,100.5)

#prints hello string at 100.5,100.5
print ("Hello",p)
```

Listing 8: "PrintXY"

Part III
Language Reference Manual

4 Lexical Conventions

The lexical structure consists of the set of basic rules that define how to write programs in plOtter. The normal token types are identifiers, keywords, operators, delimiters, and literals, as covered in the following sections.

4.1 Identifiers

An identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter (A to Z or a to z) or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9). plOtter is case sensitive, lowercase and uppercase letters are distinct.

4.2 Keywords

num	string	bool	point
true	false	while	for
if	else	return	end
fn	list		

4.3 Comments

Both single line and multi-line comments are supported.

Single Line	#
Multi Line	/* */

The pound or hash-tag symbol # is used for comment. Everything from the # to the end of the line is ignored. It can be used for either single line comment multiple line comments as follow:

```
#This is a single line comment
```

/* ... */ is used for multi line comments. Multi-line comments cannot be nested.

```
/* this is
an example of
multi-line comment */
```

4.4 Constants

false: The false value of the bool type.
true: The true value of the bool type.

4.5 White space

- Blank lines
  A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated).

- Line statement
  In plOtter, end of a statement is marked by a newline character

- Indentations
  plOtter provides no braces to indicate blocks of code for function definitions or flow control. Blocks of code are denoted by line indentation, which must be ended by end. For example:

```
if true:
        print "True"
end
else:
        print "False"
end
```

5 Types

Our language supports the following data-types:

1. Primitive types

num	32 bit floating point number
bool	Boolean value

2. Non-Primitive or object types

point	A bi-element tuple of type *num*
list	List of *num* or *string* elements
hash	Dictionary with *num* or *string* keys
string	Chain of characters enclosed by double quotes

Note: We do not have a primitive type 'char', and an index of a string points to a string of length one.

5.1 Operators

Operators are listed in order of precedence. All operators associate left to right, except for assignment, which associates right to left.

Arithmetic	+, -, *, /, %, **
Comparison	==, ! =, <, >, <=, >=, and, or
Assignment	=, + =, − =, ∗ =, / =, % =, ∗ =

6 Expressions

An expression is a combination of values, variables, and operators.

6.1 num

num a
a = 10
num b = 10

6.2 bool

bool a
a = *false*
bool b = *false*

6.3 string

string a
a = "plOtter"
string b = "plOtter"

6.4 list

Lists support both lists of num and lists of string. the list is written as a list of comma seperated elements inside square brackets. Each element of the list has an index with which it can be accessed directly. The list supports access, appending, deleting. The size can change dynamically during the course of the program with appending or deletion of an element in the list. Some examples of list initializations are given below.

list a = [1, 2, 3, 4, 5]
list b = [1.0, 2.0, 3.0, 4.0, 5.0]
list c = ["string 1","String 2","String 3"]
list a = []

Operations

append(x) : Appends the given element, x, to the end of the list. x should be of the same type as the existing elements in list. If this is the first element then that is set as the type for that list. And once this type is fixed it cant be changed after that. i.e After filling a list with num type, we cant pop all elements and insert string. That would throw an error.

> *a.append(6)*
> *will make a to [1, 2, 3, 4, 5, 6]*

pop() : this operations pops the last element of the list. If there are no elements present, it

doesnt do anything.

a.pop()
will make a to [1, 2, 3, 4, 5]

remove(x) : This operation is to remove the element at the x'th poisition. And the index starts with 0.

a.remove(3)
will remove index 3 element '4' and make a to [1, 2, 3, 5]

at(x) or "[x]" : The elements in the list can be accessed by their index using 'at' and providing the index, or by directly providing the index at square brackets. This with return the element at the index x.

a.at(2) or a[2]
will return 3

length() : This returns the length of the list at that point.

a.length() with return 4

Also the type of the elements in the list need not be specified during the initialization. The language automatically determines the type. However all the elements in a single list variable should be of same type. It should eiether be all string or all num.

6.5 point

point pMax = (10, 10)

6.6 Arithmetric operators:

2 + 3
5
num a = 5.0
num b = 3.0
a * b
15.0
14 % 3
2
22.0 12.0
1.8333333333333333

6.7 Relational and logical operators:

True or False
True

True and False
False

1 < 2
True

23 == 45
False

34 != 323
True

7 Statements

7.1 Declaration & Assignment

Declaration statements are to declare variables to use in the program. Assignment statements are used to declare variables and initialize a value to a variable, or assign a value to an initialized variable. They can be expressed in the following ways:

1. *Type Name Variable Name*
   Variable Name = *Value*

   For example:
   num x = 3
   string s = "hello"

Note: Strings values are expressed by enclosing characters in double quotes.

2. *Type Name Variable Name*
   Variable Name = *Value1*
   Variable Name = *Value2*

   For example:
   num x = 3
   x = 4.5

Note: An un-initialized variable will be initialized to the following:

num	0
string	null
point	(0,0)
list	[ ]
hash	{ }

7.2 Return

The return statement is used to exit a function, and it may optionally pass back an expression to the caller. If required, the expression is converted, by assignment, to the type of the function in which it appears. A return statement with no arguments is the same as return None. It can be expressed as:

1. *return expression*
2. *return*

8 Control Statements

8.1 Conditions

The 2 forms of conditional statements are:

- *if- else*
  This is of the form:

  if expression :
  statements
  else :
  statements
  end

  Note: The end indicates the end of the conditional block.

The expression is evaluated, and if the condition is satisfied, the set of statements associated in the block is executed. If none of the expressions' result is satisfied, the alternate set of statements associated with the *else* block are executed.

8.2 Loops

- *while*

  This has the form:

  while expression :
  statements
  end

  The expression is evaluated, and if the condition is satisfied, the set of statements

associated in the block are executed. The test takes place before each iteration of execution of the statements, and the statements are executed repeatedly, as long as the value of the expression remains satisfied.

- *for*

  This has the following form:

  - *for Initialize ; Condition ; Increment/ Decrement :*
    Statements
    end
    Initialization specifies the initialization for the loop. Condition specifies an expression to test if the loop should continue. The loop is exited when the expression is satisfied. The final part is typically an increment or decrement operation to be made on the loop variable.

  - *for Type name = start ; end :*
    Statements
    end
    Here, start specifies the value where from to begin loop iteration. End is the final part, i.e. till when the loop should continue. The value of the loop variable will increment or decrement by 1, till 'end' is reached.

9 Scope

We have broadly 3 categories for scope of variables:

- Local Scope
  These are for variables defined outside a function or control statement block, that are accessible throughout the program. They exist throughout the lifetime of the program and can be accessed inside any function or control statement block.

- Function parameters
  Formal parameters of a function are treated as local variables.

10 Functions

Functions allow structure programs as segments of code to perform tasks that also can be reused.

10.1 Function Declaration & Definition

Function declaration is done in the following format:

fn name of the function (arguments):

end

Here, arguments can be a set of arguments or no arguments. The function definition does not convey anything about the return type of the function. Some examples are given below.

Examples :

Function to convert Celsius to Fahrenheit. It takes one argument and returns Num type.

```
fn convertToCelcius(num f):
    return (f-32)/1.8
end

```

Function with no argument, returning type string

```
fn getWelcomeString():
    return "Hello World"
end
```

10.2 Function Call

Functions are called in the following way. They are basically identified by their function name and the arguments it has, to be sent. Error will be thrown if the number of arguments during the call didn't match the function signature. However defaulting the argument values can be done, in such a case the passing arguments could be less.

10.3 Built-in functions

These are the functions our language provides. However, most of these can be written by using the primitive blocks in plOtter.

Note : In all these functions and also the program all size/thickness units are *pixel* and all *color is 6 byte hex string* Eg "FFFAAF".By default color is black.Also if not specified in options default values are substituted.

line (point p1, point p2, hash options)

This function lets users draw a line from one point to another point in the SVG. It takes in the starting point, the ending point and at last options like color size etc. The options are thickness and color.

Eg. line((0,0),(10,10),"thickness":10)

printXY (point p, string s, bool dir, hash options)

This function lets users print string in the SVG. The printXY funtion, takes as argument point in the screen the point specifying the top right corner of the print, the text to be printed, boolean dir, representing whether the string should be horizontal(True) or vertical(False), and at last options like color size etc. The options are size and color.

Eg. printXY((10,10),"Hello World", true, "size":10)

We will have more functions, specific to each type of plot (For example: line plot, pie charts, etc.) similar to the barGraph function, for the user to call.

The purpose of providing these built-in functions is to demonstrate the power of our language, enabling the user to build his own custom functions, as desired. Since most of these built in functions are written in plotter

11 Sample Program

This is a sample program that gets data from a csv file, applies some options to the data and plots the data in a bar-graph. We also supply sample code for the bar-graph function, written in plOtter, which will be provided as an in-built function to the user, to show the ability to build custom graphs/plots using our language.

```
# Function written by user
list num a
a = [11,4,25.6,10,12,50,10,30,5]
barGraph(a)
```

Listing 9: Bar Graph Example

The output of the above code will be like something below.

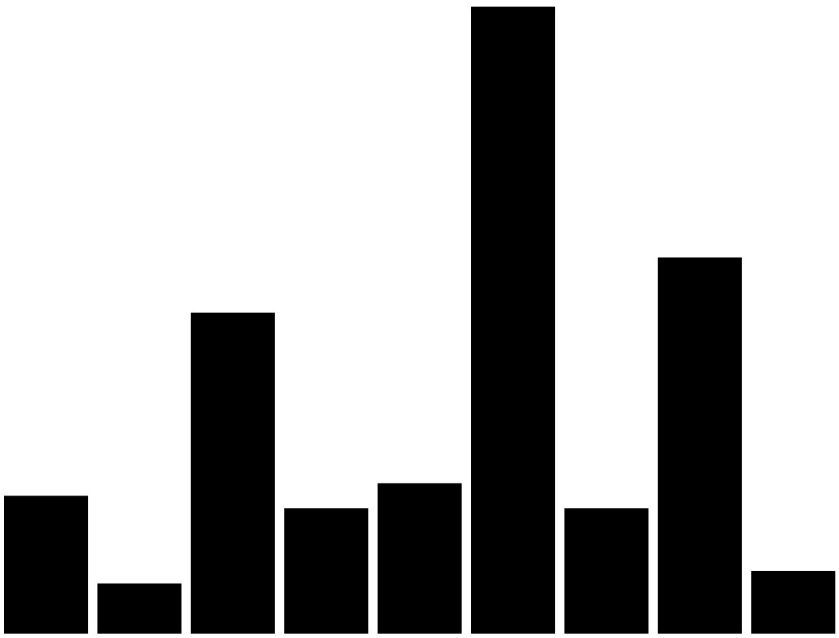


Figure 1: Simple Bar Graph

```
#Bar Graph From the Library
/*
    Method :
        barGraph(<num list var>)
```

```
5
6       Return :
7           void
8           Prints a fitting barGraph for the given data
9
10      Description :
11          The function bargraph supported as libray. Users can write their own
        too. It takes the list of num and automatically fits it in the graph based
         on the data.
12      The user just needs to call the function with the data
13
14      Future Todo :
15          Send hash, to set the properties of the graph like axis, grid, title
        etc
16  */
17  fn barGraph(list num a):
18      num maxLength
19      num maxHeight
20      num maxDataHt
21      num maxDataLn
22      num gap
23      num scaleFactor
24      num padHz
25      num padVt
26      num barWidth
27      num i
28
29
30      #Setting the dimensions of the graph
31      maxLength = 640
32      maxHeight = 480
33
34      #Max ht
35      maxDataHt = a[0]
36      for i=0;i<a.length();i=i+1:
37          if a[i] > maxDataHt:
38              maxDataHt = a[i]
39          end
40      end
41
42      #max length
43      maxDataLn = a.length()
44
45      #padding
46      padHz = 10
47      padVt = 10
48
49      #bar graph settings 10% of the graph
50      gap         = 0.1 * (maxLength - padHz) / maxDataLn
51      barWidth    = 0.9 * (maxLength - padHz) / maxDataLn
52      scaleFactor = (maxHeight - padVt) / maxDataHt
53
54      #Draw the bars, scaled and with the gap
55      num x
56      x = padHz
57      for i=0;i<a.length();i=i+1:
58          #Drawing the bar
59          rect( (x,maxHeight- a[i]*scaleFactor) , a[i]*scaleFactor , barWidth )
60          #line( (x,maxHeight- a[i]*scaleFactor) , (x,maxHeight) )
61          #print x
```

```
62          x = x + barWidth + gap
63      end
64
65
66  end
```

Listing 10: Snippet of code from Library barGraph function in plOtter.

The function barGraph is built using primitive blocks and is provided as a built-in function. However, its shown here to emphasize the power user can have for creating his own graph.

Part IV
Project Plan

We met once a week as a group, worked together every alternate day. The first two milestones, the proposal and language reference manual gave us a broad idea of the scope of our project and language semantics. Most of the language specifications were finalized by then, which was on over-estimate of what we would be able to accomplish in the semester.

Here on, we first worked to get an initial working version of the whole pipeline- from scanner, parser, abstract syntax tree, to code generation. By the time this reached completion, we reached the next milestone, which was a hello world program in our language.

From there, we incorporated a semantic checker, that maintained a symbol table and carried out scope checking and type validation. We then incrementally added features, layer by layer, going from scanner, to parser, to ast, to semantically checked ast, to target language ast, to code generation.

We also went on to set up a regression test suite that made sure the development of each feature would happen end-to-end. We added tests incrementally too, and in 2 divisions-pass tests (tests that should pass) and fail tests (tests that should not pass). Our test script would run all tests and auto-detect the kind of test, from the name of the test and report individual passed and failed tests, along with complete test statistics.

Any errors, bugs or discrepancies were resolved during the timely meetings of the system architects.

1 Project Timeline

Date	Accomplishment
Jan 25	Discuss possible languages, assign roles
Jan 27	Finalize language idea
Jan 28	Finish and submit project proposal
Feb 02	Locked down language syntax and semantics
Feb 03	Discussed the need of 2 different int and float data types
Feb 04	Finalize language data types and structures
Feb 05	Discussed the need for conditionals
Feb 07	Finalized syntax for functions
Feb 08	Compiled and submitted Language reference manual
Feb 17	Started tweaking MicroC
Feb 20	Initialized git repository
Feb 27	Added features to MicroC
Feb 29	Scrapped out MicroC and started building own compiler from scratch
Mar 4	Started the Scanner
Mar 20	Started the Parser
Mar 22	Completed an early version of Scanner, Parser and AST
Mar 24	Added Pretty print for Parser
Mar 25	Wrote sample target C++ program to write to .svg
Mar 27	Incrementally added to Scanner, Parser
Mar 31	Completed Scanner, Parser, AST
April 3	Started code generation, wrote a hello world program, wrote one test for hello world, updated the makefile
April 5	Successfully completed the pipeline, compiled a hello world program in PlOtter
April 6	Added regression test script
April 15	Added a semantic check in pipeline
April 22	Compiled an early version of semcheck
April 24	Completed pipeline with semcheck
April 26	Added line function in language
April 27	Added lists and loops
May 04	Added sample programs
May 05	Completed list operations
May 07	Added function calls
May 08	Added precise error checking
May 10	Release 1.0 of PlOtter is out and available
May 11	Final presentation, report and project files submitted Let's get some sleep.

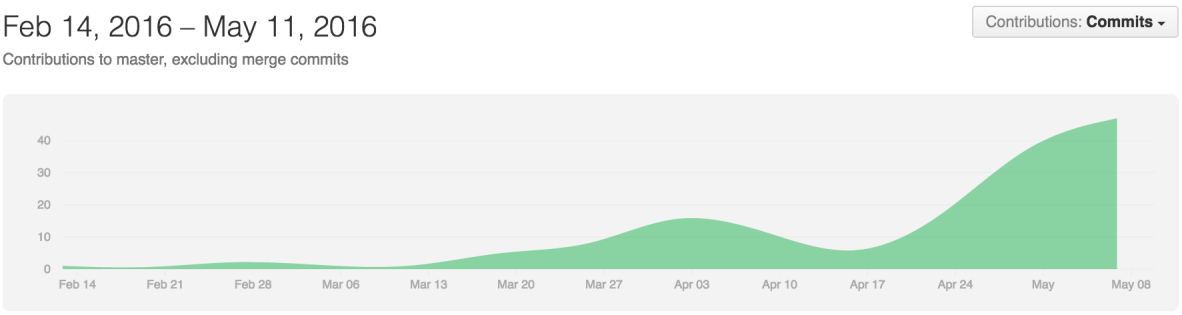
2 Development Environment

Language	OCaml, python, C++
Build System	Make
Editor	Sublime, Brackets, Vim
Version Control	Git
Bug Tracker	GitHub Issues
Operating System	OS X/ Linux

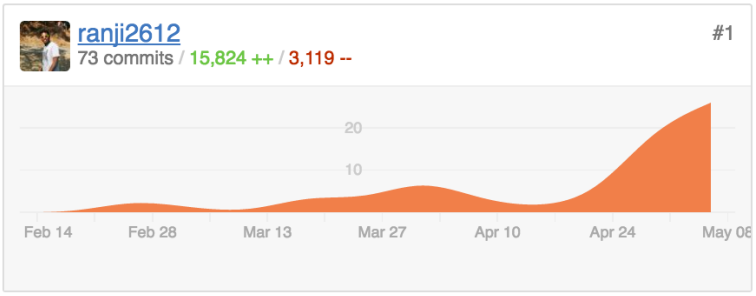
3 Version Control

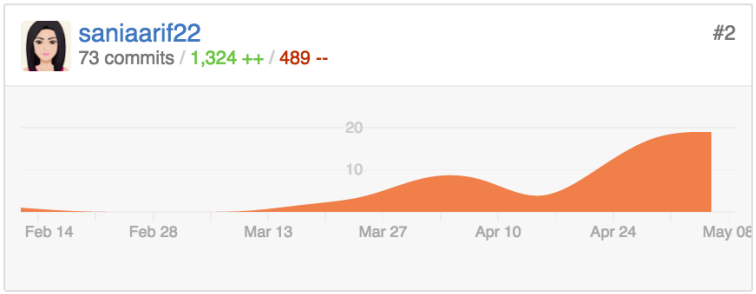
We used Git as a distributed version control system to allow all members of the group to work in collaboration and independently.
Here are some graphs produced by GitHub:

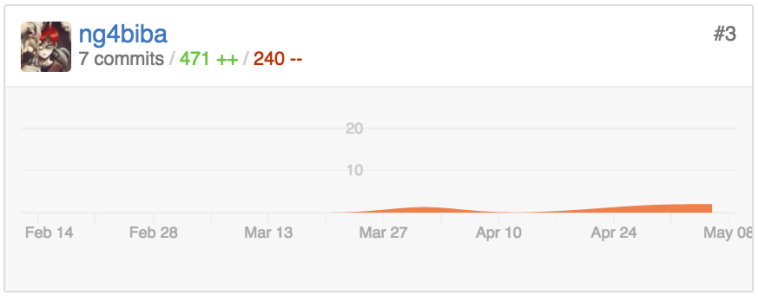
3.1 Contributions



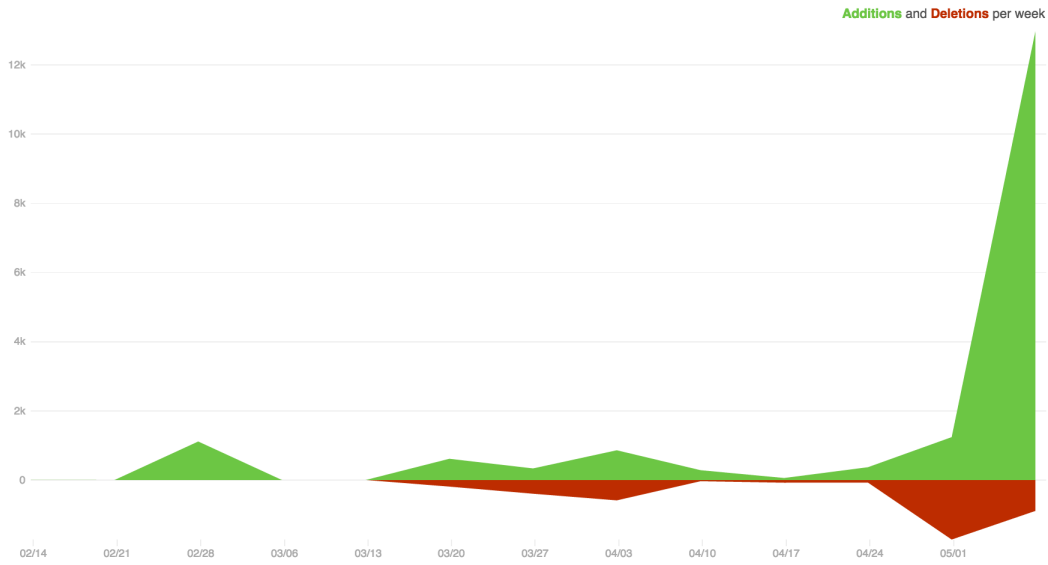
3.2 Contributors



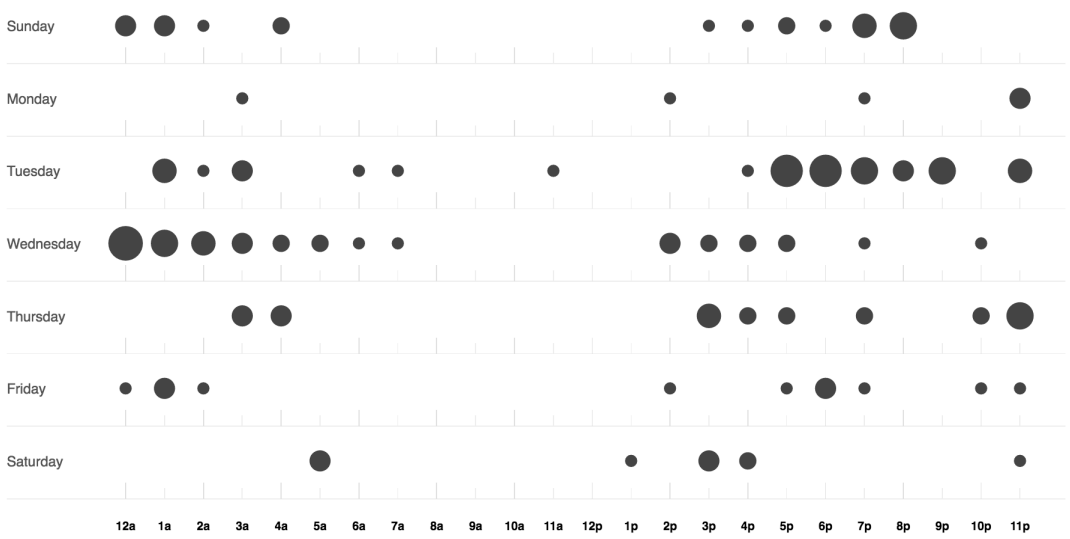




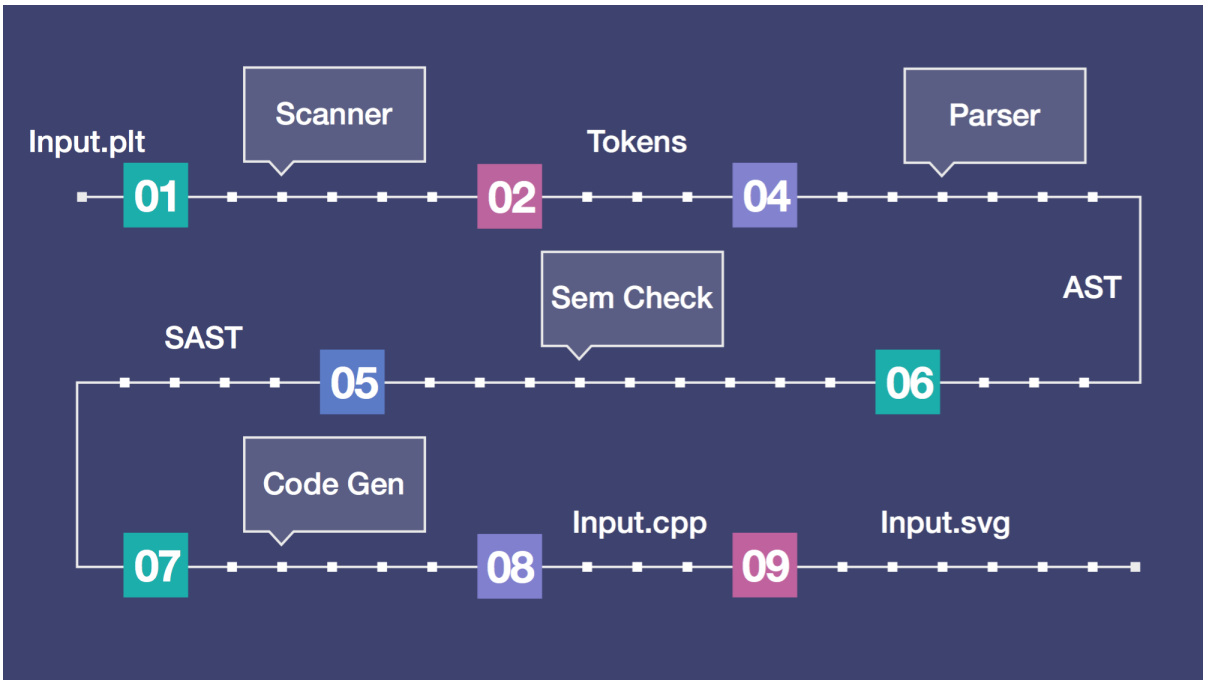
3.3 Code frequency



3.4 Punch Card



Part V
Architecture



1 High Level (plotter.ml)

This is the entry point to our compiler. It takes as input, the code in PlOtter from the lexbuffer as stdin and calls each stage of the compiler.
If an invalid input is encountered, an exception is thrown in the scanning stage and if incorrect syntax is encountered, it is reported to the user and the parser continues parsing.

This stage thus, performs error reporting and throws exceptions, exiting the program if required. It also prints out the abstract syntax tree and the semantically checked tree, depending on the compiler flags given as input.

2 Scanner (scanner.mll)

The scanner performs the following functions:

- Generates tokens, which are identifiers, operators, literals, and symbols.
- Removes single and multi-line comments
- Removes whitespaces, except for newlines, we parse as tokens.

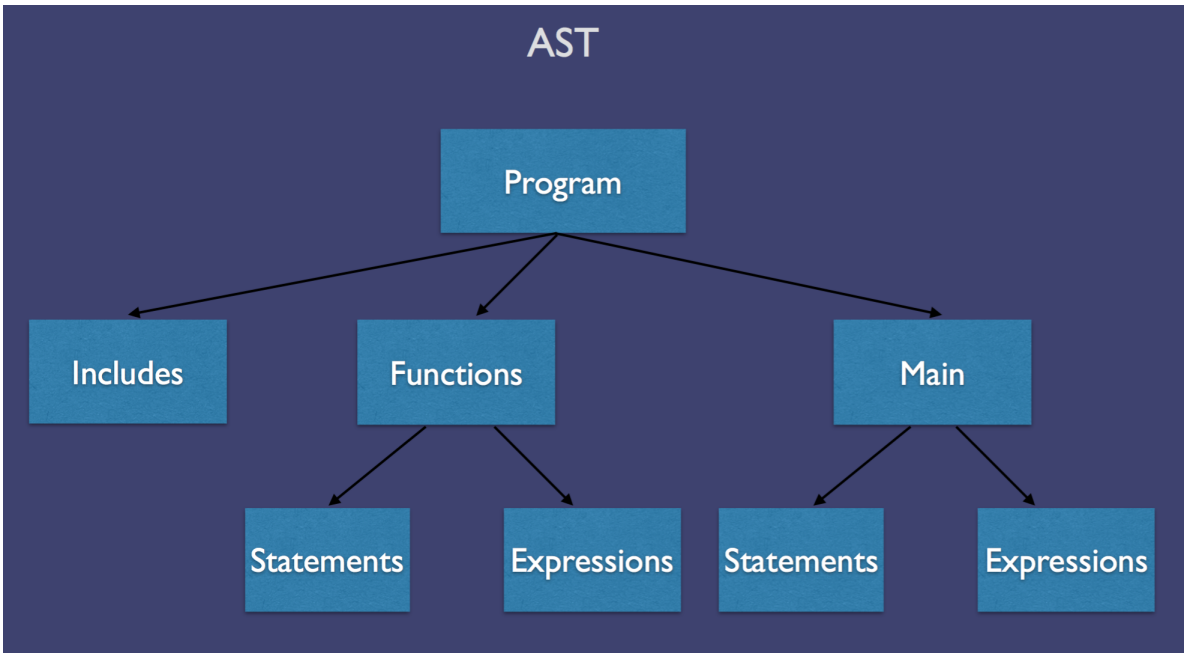
- Checks for illegal input and throws and exception for an unrecognized character along with the character and it's exact location.
  (This is our first stage of error- reporting)

3 Parser (parser.mly)

The parser takes the stream of tokens as input and generates an abstract syntax tree as output. Our parser uses the EOL (end of line whitespace token) to denote the end of a line.
Additionally, we implemented detailed and precise error checking. Every mistype or incorrect syntax is caught and displayed to the user as an erroneous entry, a possible resolution, along with the exact location of this error. (This is our second stage of error- reporting)

4 Abstract Syntax Tree (ast.ml)

The AST has the following structure:



5 Semantic Checker (semcheck.ml)

The semantic checker enforces the rules that we layed out in the Reference Manual, that includes type and scope validation and type safety. If the AST complies successfully, the semantic checker emits a semantically checked AST (SAST). We do this by maintaining a global symbol table for all statements and expressions in the main and for function names,

and different symbol tables for each user-defined function.

This is the third and last stage where an error can be caught and reported.
All compile time errors are caught and reported at semcheck. If the semcheck passes, all the stages of our compiler will pass successfully.

6 Code Generation (codegen.ml)

The code generator uses the semantically checked abstract syntax tree, passed to it by the semantic checker to construct the C++ file which contains the final code for the program, and which writes to .svg.

7 C++ Compiler

We chose to use the g++ 11 compiler. The main reason for this was the ability to use compound literals, which were essential for list and point operations.

8 Compiler Contributions

File	Authors
scanner.ml	Ibrahima, Ranjith, Sania
parser.mly	Ranjith, Sania
ast.ml	Ranjith, Sania
semcheck.ml	Ranjith, Sania
sast.ml	Ranjith, Sania
codegen.ml	Ranjith, Sania
err.ml	Sania
plotter.ml	Ranjith
cppsast.ml	Ibrahima, Ranjith, Sania
tast.ml	Ibrahima, Ranjith, Sania

Part VI
Additional Features

1 Libraries

We have built-in functions that we write in PlOtter, made available to the user as built-in functions. While these functions are not forced on the user to be used, the user can very well access then using:

include plots

```
/*
    Method :
        rect(point xy, num h, num w)

    Return :
        void

    Description :
        Prints a rectangle, from point x,y with height h and width w The function rectangle supported as libray.

    Future Todo :
        Send hash, to set the properties of the graph like axis, grid, title etc
 */
fn rect(point a, num h, num w):
    line(a,(a[0]+w,a[1]))
    line(a,(a[0],a[1]+h))
    line((a[0]+w,a[1]),(a[0]+w,a[1]+h))
    line((a[0],a[1]+h),(a[0]+w,a[1]+h))
end

fn rectFill(point a, num h, num w, string color):
    num i
    point x
    point y
    string s

    /* Make a rectanle by drawing multiple lines */
    for i=0;i<w;i=i+1:
        x = (a[0]+i, a[1])
        y = (a[0]+i, a[1]+h)
        line( x , y , color )
    end
end

/*
    Method :
        drawAxes(num xorigin, num yorigin, num maxWidth, num maxHeight)

    Return :
        void

    Description :
        Draws cross axis from the passed origin point. Takes max width and max Height to plot lines
```

```
44
45      Future Todo :
46          Send hash, to set the properties of the graph like axis, grid, title
        etc
47  */
48  fn drawAxes(num ox, num oy, num maxLength, num maxHeight):
49
50      #Draw axes
51      rect((ox,0), maxHeight, 3)
52      rect((0,maxHeight-oy),3,maxLength)
53  end
54
55  /*
56      Method :
57          barGraph(<num list var>)
58
59      Return :
60          void
61
62      Description :
63          Prints a fitting barGraph for the given data. The function bargraph
        supported as libray. Users can write their own too. It takes the list of
        num and automatically fits it in the graph based on the data.
64      The user just needs to call the function with the data
65
66      Future Todo :
67          Send hash, to set the properties of the graph like axis, grid, title
        etc
68  */
69  fn barGraph(list num a):
70      num maxLength
71      num maxHeight
72      num maxDataHt
73      num maxDataLn
74      num gap
75      num scaleFactor
76      num padHz
77      num padVt
78      num barWidth
79      num i
80
81
82      #Setting the dimensions of the graph
83      maxLength = 640
84      maxHeight = 480
85
86      #Max ht
87      maxDataHt = a[0]
88      for i=0;i<a.length();i=i+1:
89          if a[i] > maxDataHt:
90              maxDataHt = a[i]
91          end
92      end
93
94      #max length
95      maxDataLn = a.length()
96
97      #padding
98      padHz = 10
99      padVt = 10
```

```
100
101     #bar graph settings 10% of the graph
102     gap         = 0.1 * (maxLength - padHz) / maxDataLn
103     barWidth    = 0.9 * (maxLength - padHz) / maxDataLn
104     scaleFactor = (maxHeight - padVt) / maxDataHt
105
106     #Draw the bars, scaled and with the gap
107     num x
108     x = padHz
109     for i=0;i<a.length();i=i+1:
110         #Drawing the bar
111         rect( (x,maxHeight- a[i]*scaleFactor) , a[i]*scaleFactor , barWidth )
112         #line( (x,maxHeight- a[i]*scaleFactor) , (x,maxHeight) )
113         #print x
114         x = x + barWidth + gap
115     end
116
117
118 end
119
120 /*
121     Method :
122         lineGraph(num xorigin, num yorigin, num maxWidth, num maxHeight)
123
124     Return :
125         void
126
127     Description :
128         Draws a line graph  based on the points give. To draw axes call it
    seperately
129
130     Future Todo :
131         Send hash, to set the properties of the graph like axis, grid, title
    etc
132 */
133 fn lineGraph(list num a):
134     num maxLength
135     num maxHeight
136     num maxDataHt
137     num maxDataLn
138     num gap
139     num scaleFactor
140     num padHz
141     num padVt
142     point p1
143     point p2
144     num i
145
146
147     #Setting the dimensions of the graph
148     maxLength = 640
149     maxHeight = 480
150
151     #Max ht
152     maxDataHt = a[0]
153     for i=0;i<a.length();i=i+1:
154         if a[i] > maxDataHt:
155             maxDataHt = a[i]
156         end
157     end
```

```
158
159     #max length
160     maxDataLn = a.length()
161
162     #padding
163     padHz = 10
164     padVt = 10
165
166     #Line graph settings 10% of the graph
167     gap         = (maxLength - padHz) / maxDataLn
168     scaleFactor = (maxHeight - padVt) / maxDataHt
169
170     #Draw the bars, scaled and with the gap
171     num x
172     num j
173     num y1
174     num y2
175     x = padHz
176
177     if a.length() > 0:
178
179         for i=1;i<a.length();i=i+1:
180             line( (x , maxHeight - a[ i - 1 ] * scaleFactor) , ( x + gap ,
        maxHeight- a[ i ]*scaleFactor) )
181
182             #dots on the data points
183             rect ( (x - 2.5, maxHeight - a[ i - 1 ] * scaleFactor - 2.5) , 5
        , 5)
184             rect ( (x - 2.5, maxHeight - a[ i - 1 ] * scaleFactor - 2.5) , 5
        , 5)
185             x = x + gap
186
187         end
188     end
189 end
```

2 Customizability

Alternatively, the user can implement their own functions using our most basic, primitive functions print and line. Some of the features are **rect(point p, num width, num height)**

This draws a rectangle of width and height, starting from p, with options. The options we are considering now are simple like, thickness(of the border), color(of the border) and fill(boolean variable specifying fill should be done). If fill is true, additional option called fillColor is also required, if not specified default of black fill is used.

Eg. rectangle((10,10),5, 10,"thickness":1, "color":"FFFFFF","fill":false)

rectFill(point p, num width, num height)

This draws a rectangle of width and height, starting from p, with options. The options we are considering now are simple like, thickness(of the border), color(of the border) and fill(boolean variable specifying fill should be done). If fill is true, additional option called fillColor is also required, if not specified default of black fill is used.

Eg. rectangle((10,10),5, 10,"thickness":1, "color":"FFFFFF","fill":false)

barGraph(list data)

This draws a barGraph given the data and specified options. The options include, num max-width, num max-height, num color (of the bar), xaxis(boolean representing whether

the axis should be there), bool yaxis, string xLabel, string yLabel, bool grid(to have/not have grid lines) and string title (For example: padding, max width, max height, etc.)

Eg.

hash options = {"xAxis": true,"yAxis": true,"xLabel": "Time","yLabel": "Celcius", "title":"Temperature with Time","grid":true}

barGraph([5,1,4,2,7])

lineGraph(list data, hash options)

This draws a lineGraph given the data and specified options. The options include, num max-width, num max-height, num color (of the bar), xaxis(boolean representing whether the axis should be there), bool yaxis, string xLabel, string yLabel, bool grid(to have/not have grid lines) and string title (For example: padding, max width, max height, etc.). ALthough the options feature is not there now, it'll be updated further.

Eg.

hash options = {"xAxis": true,"yAxis": true,"xLabel": "Time","yLabel": "Celcius", "title":"Temperature with Time","grid":true}

barGraph([5,1,4,2,7])

Part VII
Test Plan

1 Test Automation

Our testing automation program is invoked by calling make test. Our script either takes as input, specific files to test on, or iterates through each file in the test directory ending that is ending in .plt.
We define 2 kinds of tests:

- Pass tests- Tests that are supposed to pass
- Fail tests- Tests that are supposed to fail

After running the program, the script first detects the type of test (pass test vs fail test), and if the test was supposed to pass, it compares the generated result (a .svg file) with the corresponding .svg file. It then maintains a list of the files that pass tests and the files that fail and generates statstics at the end.

2 Test Cases

Our test directory contains all of our tests. The pass tests are preceeded by the word 'pass' and the fail tests are preceeded by the word 'fail'. As of commit bdb5887, we have 85 tests. We have tests for the following categories:
1. Assignment
2. Operators
3. Comments
4. Printing
5. Loops
6. Conditionals
7. Lists
8. List operations
9. Functions
10. Libraries
11. Scope

3 Testing Roles

The test script was written by Ranjith. Tests were written by either Sania or Ranjith, the person who implemented each feature. We also raised issues for each bug as we encountered it and were responsible collectively for closing an issue on resolution.

Part VIII
Future Work

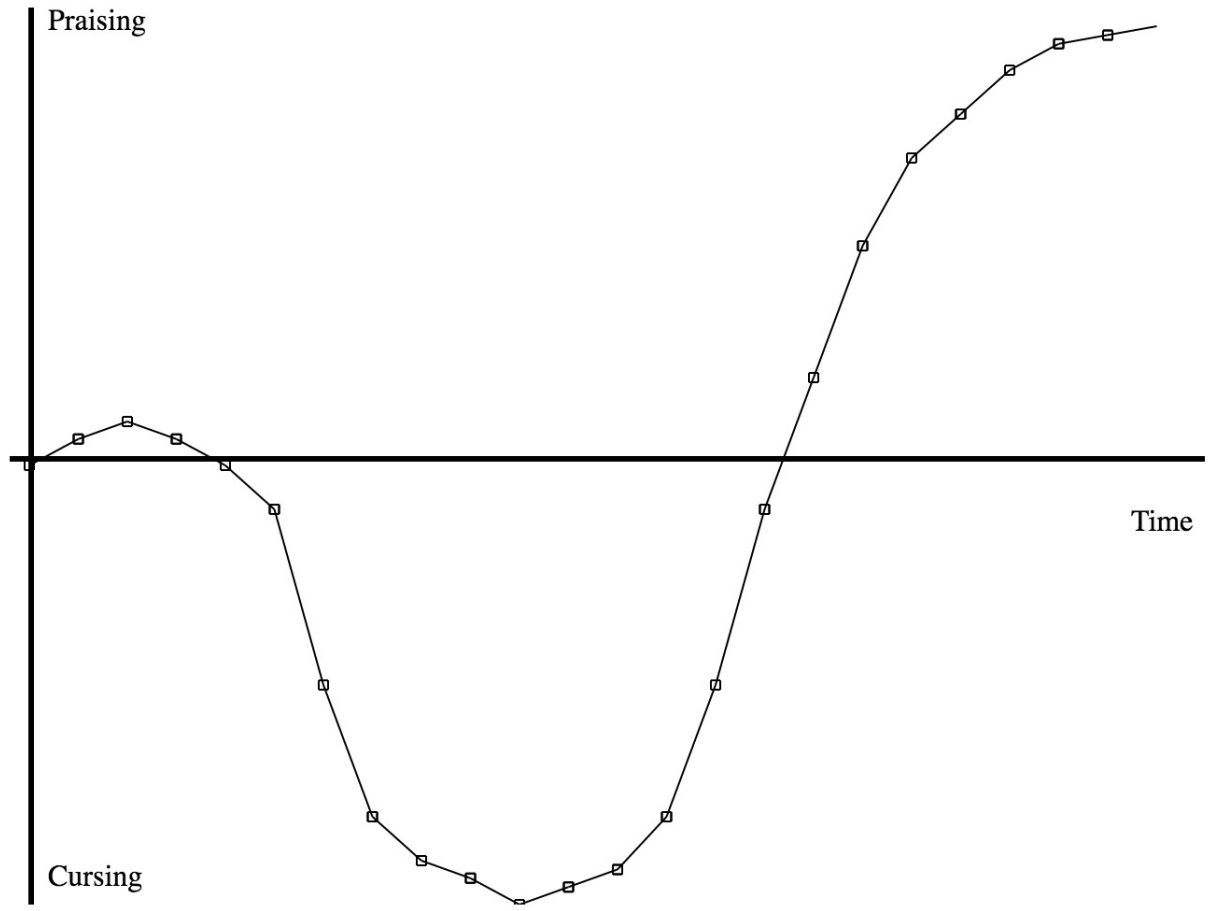
- More customizability
  Adding more features such as line type, line width and weight, background color, etc.
- Support for math functions
  Such as square, cube, sine, cosine, etc.
- Import Data
  Ability to import data from additional sources (eg: csv)
- More responsive REPL window
- More libraries

Part IX
Lessons Learned

1 Sania & Ranjith Kumar

- Pair Programming saves lives.
- Use Prof. Edwards' slides.
- Choose teammates wisely, you'll be stuck with them for the term. Courtesy: Prof. Edwards
- OCaml is awesome, give it time.
  Our graph of appreciating OCaml:

Output SVG



2 Ibrahima

Learn Ocaml and get to familiarize with the syntax as early as possible. Understanding the architecture of the compiler and how its pieces fit together is a must in order to keep the progress going. Lastly group project is amazing,specially when you feel included in the work, when members work with one another and shared their knowledge with the rest of the team.

Part X
Appendix

./src/.gitignore

```
tests/*.cpp
 tests/*.svg
 programs/*.cpp


```

./src/ast.ml

```
(* operations *)
type  ops =
        | Add | Sub | Mul  | Div | Mod
        | Equal | Neq | Less | Leq | Greater | Geq
        | And | Or
        | Square

type bool =
        | True | False


(*expressions*)
type expr =
    Literal_Num of float
  | Literal_Str of string
  | Point of expr * expr
  | Literal_List of expr list                (* Eg [ expr, expr, .. ] *)
  | Binop of expr * ops * expr               (* Binary Ops *)
  | Id of string                             (* identifiers *)
  | Bool of bool                             (* True *)
  | Length of expr                           (* a.length() *)
  | Access of expr * expr                    (* a.at(3), a[3] *)


type stmt = (* Statements *)
    Expr of expr
  | Var_Decl of string * string              (* (type, id) *)
  | List_Decl of string * string
  | Passign of expr * expr * stmt                  (* (type, p1, p2) *)
  | Assign of expr * expr                    (* a = 2 *)
  | Append of expr * expr                    (* a.append(7) *)
  | Pop of expr                              (* a.pop() *)
  | Remove of expr * expr                    (* a.remove(3) *)
  | Fcall  of string * expr list             (* a.() *)
  | PrintXY of expr * expr                   (* printXY ( 5 , (1,2)) *)
  | Print of expr                            (* print 5 *)
  | LineVar of expr * expr                   (* line(p,q) *)
  | LineVarColor of expr * expr * expr       (* line(p,q,"blue") *)
  | LineRaw of expr * expr * expr * expr     (* line((3,4), (7,9)) *)
  | LinePX of expr * expr * expr             (* line((3,4), x) , line(x,
    (3,4)) *)
  | For of stmt * expr * stmt * stmt list    (* for i=0; i<5; i=i+1: *)
  | While  of expr * stmt list
  | Ifelse of expr * stmt list * stmt list
  | Return of expr
  | Noexpr
  | Fdecl of fdecl and
      fdecl = {
       fname : string;
       args  : stmt list;
       body  : stmt list;
      }


type program = {
                    funcs : stmt list;
                    main  : stmt list;
```

```
59                }
60
61
62
63  (* Pretty Print *)
64
65  let rec string_of_expr = function
66      Literal_Num(l) -> string_of_float l ^ "0"
67    | Literal_Str(l) -> l
68    | Point(e1, e2) -> "(" ^ string_of_expr e1 ^ "," ^ string_of_expr e2 ^ ")"
69    | Literal_List(l) -> "[" ^ (String.concat "," (List.map string_of_expr l))
      ^ "]"
70    | Id(s) -> s
71    | Binop(e1, o, e2) ->
72        string_of_expr e1 ^ " " ^
73        (match o with
74   Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/"
75        | Equal -> "==" | Neq -> "!="
76        | Mod -> "%"
77        | And -> "&&" | Or ->"||"
78        | Square -> "**"
79        | Less -> "<" | Leq -> "<="
80        | Greater -> ">" | Geq -> ">="
81        ) ^ " " ^ string_of_expr e2
82    | Bool(x) -> if x = True then "true" else "false"
83    | Length(v) -> string_of_expr v ^ ".length()\n"
84    | Access(v, e) -> string_of_expr v ^ ".at(" ^ ( string_of_expr e ) ^ ")\n"
85
86
87
88
89  let rec string_of_stmt = function
90      Expr(expr) -> string_of_expr expr ^ ""
91    | Var_Decl(tp, id) -> tp ^ " " ^ id ^ "\n"
92    | List_Decl(tp, id) -> "list " ^ tp ^ " " ^ id ^ "\n"
93    | Passign(v, e1, e) -> " " ^ string_of_expr v ^ " = " ^ ( string_of_expr
      e1 ) ^ "\n"
94    | Assign(v, e) -> "" ^ string_of_expr v ^ " = " ^ ( string_of_expr e )
95    | Append(v, e) -> string_of_expr v ^ ".append(" ^ ( string_of_expr e ) ^ "
      )\n"
96    | Pop(v) -> string_of_expr v ^ ".pop()\n"
97    | Remove(v, e) -> string_of_expr v ^ ".remove(" ^ ( string_of_expr e ) ^ "
      )\n"
98    | Fcall(v, el) ->  v ^ "("^ (String.concat "," (List.map string_of_expr el
      )) ^")\n"
99    | PrintXY(e1,e2) -> "printXY( " ^ string_of_expr e1 ^ "," ^ string_of_expr
        e2 ^ ")\n"
100   | Print(e) -> "print " ^ string_of_expr e ^ "\n"
101   | LineVar(e1,e2)-> "line (" ^ string_of_expr e1 ^ "," ^ string_of_expr e2
      ^ ")" ^ "\n"
102   | LineVarColor(e1,e2, c)-> "line (" ^ string_of_expr e1 ^ "," ^
      string_of_expr e2 ^ "," ^ string_of_expr c ^ ")" ^ "\n"
103   | LineRaw(e1,e2,e3,e4)-> "line ( (" ^ string_of_expr e1 ^ "," ^
      string_of_expr e2 ^ ")" ^ "," ^ "(" ^ string_of_expr e3
104                             ^ "," ^ string_of_expr e4 ^ ") )\n"
105   | LinePX(e1, e2, e3)-> "line ( ( " ^ string_of_expr e1 ^ "," ^
      string_of_expr e2 ^ ") ," ^ string_of_expr e3 ^ ") \n"
106   | For(s1, e1, s2, body) -> "for " ^ string_of_stmt s1 ^ " ; " ^
      string_of_expr e1 ^ " ; " ^ string_of_stmt s2 ^ ": \n"
107                           ^ ( String.concat "\n\t" (List.map
```

```
    string_of_stmt body) )
                                ^ "\nend\n"
  | While(e, body) -> "while " ^ string_of_expr e ^ " :\n" ^ (String.concat
    "\n\t" (List.map string_of_stmt body)) ^ "\nend\n"
  | Ifelse(e, succ_stmt, else_stmt) -> "if " ^ string_of_expr e ^ " :\n" ^ (
    String.concat "\n\t" (List.map string_of_stmt succ_stmt)) ^ "\nelse:\n" ^
    (String.concat "\n\t" (List.map string_of_stmt else_stmt)) ^ "end\n"
  | Return(expr) -> "return " ^ string_of_expr expr ^ "\n"
  | Noexpr -> ""
  | Fdecl(f) -> string_of_fdecl f and
  string_of_fdecl fdecl =
      "fn " ^ fdecl.fname ^ "(" ^
        ( String.concat ", " (List.map (fun s -> string_of_stmt s) fdecl.
    args) ) ^
        "):\n" ^
      ( String.concat "" (List.map string_of_stmt fdecl.body) ) ^
      "\nend\n"

let string_of_program prog =
  String.concat "\n" (List.map string_of_stmt prog.funcs)
  ^ "\n----------\n" ^
  String.concat "\n" (List.map string_of_stmt prog.main)
```

./src/codegen.ml

```
open Ast
 open Semcheck

 let convert prog =
   check prog.funcs;
   check prog.main;
   let rec create_expr = function
       | Ast.Literal_Num(l) -> (string_of_float l) ^ "0"
       | Ast.Literal_Str(l) -> l
       | Ast.Point(e1,e2) -> "(float[2]){(float)(" ^ create_expr e1^ "),(float)(" ^ create_expr e2 ^ ")}"
       | Ast.Literal_List(l)-> "{" ^ (String.concat "," (List.map create_expr l) ) ^ "}"
       | Ast.Id(s) -> "(" ^ s ^ ")"
       | Ast.Binop(e1, o, e2) ->
           "(" ^ create_expr e1 ^ " " ^
           (match o with
    Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/"
         | Equal -> "==" | Neq -> "!="
         | Mod -> "%"
         | And -> "&&" | Or ->"||"
         | Square -> "**"
         | Less -> "<" | Leq -> "<="
         | Greater -> ">" | Geq -> ">="
         ) ^ " " ^ create_expr e2 ^ ")"
       | Ast.Bool(x) -> if x = True then "true" else "false"
       | Ast.Length(v) -> create_expr v ^ ".size()"
       | Ast.Access(v,e) -> create_expr v ^ "[int(" ^ ( create_expr e ) ^ ")]
    "



    in
    let printAssign v e = match e with
         | Ast.Point(e1,e2) -> "__inbuildt_cp(" ^ create_expr v ^ ", " ^ create_expr e ^ ");"
         | _ -> create_expr v ^ " = " ^ ( create_expr e ) ^ ";"
    in
    let printFunArgs = function
        | Ast.Var_Decl(tp, id) ->
            (match tp with
                  "num" -> "float" ^ " " ^ id
                | "string" -> "string" ^ " " ^ id
                | "point" -> "float" ^ " " ^ id ^ "[2]"
                | _ -> "bool" ^ " " ^ id
            )
        | Ast.List_Decl(tp, id) ->
            (match tp with
                  "num" -> "vector <float>" ^ " & " ^ id
                | "string" -> "vector <string>" ^ " & " ^ id
                | "point" -> "vector <array<float , 2>>" ^ " & " ^ id
                | _ -> "vector <bool>" ^ " & " ^ id
            )
         | _ -> raise (Failure "Its not possible :P ")
    in
    let remSemColon s =
         Bytes.set s (String.length s - 1) ' '; s
    in

```

```
56      let rec create_stmt = function
57           | Ast.Expr(expr) -> create_expr expr
58           | Ast.Var_Decl(tp, id) ->
59             (match tp with
60                   "num" -> "float" ^ " " ^ id ^ ";\n"
61                 | "string" -> "string" ^ " " ^ id ^ ";\n"
62                 | "point" -> "float" ^ " " ^ id ^ "[2];\n"
63                 | _ -> "bool" ^ " " ^ id ^ ";\n"
64             )
65         | Ast.List_Decl(tp, id) ->
66             (match tp with
67
68                   "num" -> "vector <float>" ^ " " ^ id ^ ";\n"
69                 | "string" -> "vector <string>" ^ " " ^ id ^ ";\n"
70                 | "point" -> "vector <array<float , 2>>" ^ " " ^ id ^ ";\n"
71                 | _ -> "vector <bool>" ^ " " ^ id ^ ";\n"
72             )
73           | Ast.Passign(v, e, e2) ->
74             (* Setting the point elements seperately *)
75             "__inbuildt_cp(" ^ create_expr v ^ ", (float[2])" ^ create_expr
    e ^ ")"
76           | Ast.Assign(v, e) -> printAssign v e
77           | Ast.Append(v, e) -> create_expr v ^ ".push_back(" ^ ( create_expr
    e ) ^ ");\n"
78           | Ast.Pop(v) -> create_expr v ^ ".pop_back();\n"
79           | Ast.Remove(v,e) -> create_expr v ^ ".erase(" ^ (create_expr v) ^ "
    .begin() + " ^ ( create_expr e ) ^ ");\n"
80           | Ast.PrintXY(e1,e2) -> "put_in_svg( " ^ create_expr e1 ^ "," ^
    create_expr e2 ^");\n"
81           | Ast.Print(e) -> "put_in_svg( " ^ create_expr e ^ ");\n"
82         | Ast.LineVar(e1, e2) -> "put_in_svg (" ^ create_expr e1 ^ "," ^
    create_expr e2 ^");\n"
83         | Ast.LineVarColor(e1, e2, c) -> "put_in_svg (" ^ create_expr e1 ^ ",
    " ^ create_expr e2 ^", " ^ create_expr  c ^");\n"
84         | Ast.LineRaw(e1, e2, e3, e4) -> "put_in_svg (" ^ create_expr e1 ^ ",
    " ^ create_expr e2
85                                          ^ "," ^ create_expr e3 ^ "," ^
    create_expr e4 ^");\n"
86         | Ast.LinePX(e1, e2, e3) -> "put_in_svg (" ^ create_expr e1 ^ "," ^
    create_expr e2
87                                          ^ "," ^ create_expr e3 ^ ");\n"
88         | Ast.For(s1, e1, s2, body) -> "for (" ^ create_stmt s1 ^ " " ^
    create_expr e1 ^ " ; "
89                                          ^ ( remSemColon (create_stmt s2 )) ^ "
     ) { \n"
90                                          ^ String.concat "" (List.map
    create_stmt body) ^ "\n } \n"
91         | Ast.While(e, body) -> "while " ^ create_expr e ^ " { \n" ^ String.
    concat "" (List.map create_stmt body) ^ "}\n"
92         | Ast.Ifelse(e, s1, s2) -> "if " ^ create_expr e ^ " { \n" ^ String.
    concat "" (List.map create_stmt s1)
93             ^ "} else { \n" ^ String.concat "" (List.map create_stmt s2) ^ "
    }\n"
94           | Ast.Return(expr) -> "return " ^ create_expr expr ^ ";\n"
95           | Ast.Noexpr       -> ""
96           | Ast.Fcall(v,el)  -> v ^ "("^ (String.concat "," ( List.map (fun s
    ->create_expr s) el)) ^");"
97           | Ast.Fdecl(f)     -> string_of_fdecl f and
98                string_of_fdecl fdecl =
99                   "void " ^ fdecl.fname ^ "(" ^
```

```
                    ( String.concat ", " (List.map (fun s -> printFunArgs s)
    fdecl.args) ) ^
                     "){\n" ^
                   ( String.concat "" (List.map create_stmt fdecl.body) ) ^
                   "\n}\n"

  in

   "#include <iostream>\n#include <fstream>\n#include <vector>\n" ^
   "using namespace std;\n"^

   "ofstream f;\n"^
   "// SVG content\n"^
   "void __inbuildt_cp(float a[2], float b[2]) {" ^
"a[0] = *b;" ^
"a[1] = *(b+1); " ^
   "}" ^
   "void put_in_svg(float p1[])\n" ^
   "{  \n" ^
   "    f << \"<text x='250' y='150'>\\n\";\n"^
   "    f << p1[0] << \"  \" << p1[1];\n" ^
   "    f << \"\\n</text>\\n\";\n"^
   "}\n" ^
   "void put_in_svg(float p1[], float p2[], string color=\"black\")\n"^
   "{"^
   "  f << \"<line x1='\" + to_string(p1[0]) + \"' y1='\"+  to_string(p1
    [1])+\"' x2='\"+ to_string(p2[0]) +\"' y2='\"+ to_string(p2[1]) +\"' style
    ='stroke:\" + color+\";stroke-width:1'/>\\n\"; \n" ^
   "}\n"^

   "void put_in_svg(float p1[], float p2, float p3)\n"^
   "{"^
   "  f << \"<line x1='\" + to_string(p1[0]) + \"' y1='\"+  to_string(p1
    [1])+\"' x2='\"+ to_string(p2) +\"' y2='\"+ to_string(p3) +\"' style='
    stroke:rgb(0,0,0);stroke-width:1'/>\\n\"; \n" ^
   "}\n"^

   "void put_in_svg(float p1, float p2, float p3[])\n"^
   "{"^
   "  f << \"<line x1='\" + to_string(p1) + \"' y1='\"+  to_string(p2)+\"'
    x2='\"+ to_string(p3[0]) +\"' y2='\"+ to_string(p3[1]) +\"' style='stroke:
    rgb(0,0,0);stroke-width:1'/>\\n\"; \n" ^
   "}\n"^

   "void put_in_svg(float x1, float y1, float x2, float y2)\n"^
   "{"^
   "  f << \"<line x1='\" + to_string(x1) + \"' y1='\"+  to_string(y1)+\"'
    x2='\"+ to_string(x2) +\"' y2='\"+ to_string(y2) +\"' style='stroke:rgb
    (0,0,0);stroke-width:1'/>\\n\"; \n" ^
   "}\n"^

   "void put_in_svg(std::string content)\n"^
   "{\n"^
   "  f << \"<text x='250' y='150'>\\n\";\n"^
   "  f << content;\n"^
   "  f << \"\\n</text>\\n\";\n"^
   "}\n"^

  "void put_in_svg(std::string content, float p[2])\n"^
   "{\n"^
```

```
151      "  f << \"<text x='\" + to_string(p[0]) + \"' y='\"+  to_string(p[1])
        +\"' >\\n\";\n"^
152      "  f << content;\n"^
153      "  f << \"\\n</text>\\n\";\n"^
154      "}\n"^
155
156      "void put_in_svg(float content)\n"^
157      "{\n"^
158      "  f << \"<text x='250' y='150'>\\n\";\n"^
159      "  f << content;\n"^
160      "  f << \"\\n</text>\\n\";\n"^
161      "}\n"^
162
163      "//All user & library functions goes here\n" ^
164
165      String.concat "" (List.map create_stmt prog.funcs) ^
166
167      "//Main prog starts\n"^
168
169      "int main() {\n"^
170      (* change the name to be the filename.svg based on the file which is ran
         *)
171      "  f.open (\"hello.svg\");\n"^
172
173      "  // Prolog for the SVG image\n"^
174      "  f << \"<svg xmlns=\\\"http://www.w3.org/2000/svg\\\" xmlns:xlink=\\\"
        http://www.w3.org/1999/xlink\\\" width=\\\"1024\\\" height=\\\"768\\\">\";
         \n"^
175      "  f << \"\\n\"; \n"^
176
177     String.concat "" (List.map create_stmt prog.main) ^
178
179      "  f << \"</svg>\"; \n" ^
180     "return 0;\n}\n"
181
182
```

./src/cppsast.ml

```
(* converts plotter AST to C++ AST *)
open Tast
open Ast

(* Convert ast expr to Tast expr *)
let match_of e = match e with
| Ast.Add -> Tast.Add
| Ast.Sub -> Tast.Sub
| Ast.Mul -> Tast.Mul
| Ast.Div -> Tast.Div
| Ast.Mod -> Tast.Mod
| Ast.Equal -> Tast.Equal
| Ast.Neq -> Tast.Neq
| Ast.Less -> Tast.Less
| Ast.Leq -> Tast.Leq
| Ast.Greater -> Tast.Greater
| Ast.Geq -> Tast.Geq
| Ast.And -> Tast.And
| Ast.Or -> Tast.Or
| Ast.Square -> Tast.Square

let convert_to_cppast stmts_list =
  let rec convert_to_texpr  = function
    | Ast.Literal_Num(v) -> Tast.Literal_Num(v)
    | Ast.Literal_Str(v) -> Tast.Literal_Str(v)
    | Ast.Literal_List(e) ->
      let te = List.map (fun s -> convert_to_texpr s) e in
      Tast.Literal_List(te)
    | Ast.Bool(v)        -> if v = True then Tast.Bool(True) else Tast.Bool(
   False)
    | Ast.Length(v) -> Tast.Length(convert_to_texpr v)
    | Ast.Binop(e1, op, e2) ->
        let te1 = convert_to_texpr e1 in
        let te2 = convert_to_texpr e2 in
        let top = match_of op in
        Tast.Binop(te1, top, te2)
    | Ast.Id(v)          -> Tast.Id(v)
    | Ast.Access(v,e) ->
        let texpr_acc = convert_to_texpr e in
        let tv = convert_to_texpr v in
        Tast.Access(tv, texpr_acc)
    | Ast.Point(e1, e2) ->
        let te1 = convert_to_texpr e1 in
        let te2 = convert_to_texpr e2 in
        Tast.Point(te1, te2)

  in

  (* Convert ast stmt to Tast stmt *)
  let rec convert_to_tstmt stmt = function
    | Ast.Expr(e) ->
        let texpr = convert_to_texpr e in
        Tast.Texpr(texpr)
    | Ast.Var_Decl(t,e) ->Tast.Var_Decl(t,e)
    | Ast.List_Decl(t,id) -> Tast.List_Decl(t, id)
    | Ast.Passign(v,e1,e2) -> Tast.Noexpr
    | Ast.Assign(v,e) ->
      let te1 = convert_to_texpr e in
      let tv = convert_to_texpr v in
```

```
59          Tast.Assign(tv, te1)
60      | Ast.Print(e) -> Tast.Print(convert_to_texpr e)
61      | Ast.Append(v,e) ->
62        let texpr_app = convert_to_texpr e in
63        let tv = convert_to_texpr v in
64        Tast.Append(tv, texpr_app)
65      | Ast.Remove(v,e) ->
66          let texpr_rm = convert_to_texpr e in
67          let tv = convert_to_texpr v in
68          Tast.Remove(tv, texpr_rm)
69      | Ast.Pop(e) -> Tast.Pop(convert_to_texpr e)
70      | Ast.LineVar(e1,e2) ->
71        let texpr1 = convert_to_texpr e1 in
72        let texpr2 = convert_to_texpr e2 in
73        Tast.LineVar(texpr1, texpr2)
74      | Ast.LineRaw(e1,e2,e3,e4) ->
75        let texpr1 = convert_to_texpr e1 in
76        let texpr2 = convert_to_texpr e2 in
77        let texpr3 = convert_to_texpr e3 in
78        let texpr4 = convert_to_texpr e4 in
79        Tast.LineRaw(texpr1, texpr2,texpr3,texpr4)
80      | Ast.For(s1, e1, s2, body) ->
81        let tstmt1 = convert_to_tstmt stmt s1 in
82        let texpr1 = convert_to_texpr e1 in
83        let tstmt2 = convert_to_tstmt stmt s2 in
84        let tbody = List.map (fun s -> convert_to_tstmt stmt s) body in
85        Tast.For(tstmt1, texpr1,tstmt2,tbody)
86      | Ast.While(e,body) ->
87        let texpr = convert_to_texpr e in
88        let tbody = List.map (fun s -> convert_to_tstmt stmt s) body in
89        Tast.While(texpr,tbody)
90
91      | Ast.Ifelse(e, succ_stmt, else_stmt) ->
92        let texpr = convert_to_texpr e in
93        let tsucc_stmt = List.map (fun s -> convert_to_tstmt stmt s) succ_stmt
    in
94        let telse_stmt = List.map (fun s -> convert_to_tstmt stmt s) else_stmt
    in
95        Tast.Ifelse(texpr, tsucc_stmt, telse_stmt)
96      | Ast.Return(e) -> Tast.Return(convert_to_texpr e)
97      | Ast.Fdecl(f) ->
98          let f_name = f.fname in
99          let f_args = List.map (fun s -> convert_to_tstmt stmt s) f.args in
100         let f_body = List.map (fun s -> convert_to_tstmt stmt s) f.body in
101       Tast.Fdecl({
102                 fname = f_name;
103                 args  = f_args;
104                 body  = f_body;
105                 })
106     | Ast.Fcall(f, el) ->
107       let tel = List.map (fun s -> convert_to_texpr s) el in
108       Tast.Fcall(f, tel)
109     | Ast.PrintXY (e1, e2) ->
110       let te1 = convert_to_texpr e1 in
111       let te2 = convert_to_texpr e2 in
112       Tast.PrintXY(te1, te2)
113     | Ast.LinePX(e1, e2, e3) ->
114       let te1 = convert_to_texpr e1 in
115       let te2 = convert_to_texpr e2 in
116       let te3 = convert_to_texpr e3 in
```

```
117             Tast.LinePX(te1, te2, te3)
118       | Ast. Noexpr -> Tast.Noexpr
119
120     in
121     List.map (fun s -> convert_to_tstmt s) stmts_list;
122
```

./src/err.ml

```
open Printf
 open Lexing
 open Parsing

 exception ScanError of string

 let print_position lexbuf msg =
   let start = lexeme_start_p lexbuf in
    let finish = lexeme_end_p lexbuf in
     (fprintf stderr "Line %d: char %d..%d: %s: \"%s\" \n"
     start.pos_lnum
     (start.pos_cnum - start.pos_bol)
     (finish.pos_cnum - finish.pos_bol)
     msg
     (Lexing.lexeme lexbuf))

 let lexer_from_channel fname ch =
   let lex = Lexing.from_channel ch in
   let pos = lex.lex_curr_p in
     lex.lex_curr_p <- { pos with pos_fname = fname; pos_lnum = 1; } ;
     lex

 let lexer_from_string str =
   let lex = Lexing.from_string str in
   let pos = lex.lex_curr_p in
     lex.lex_curr_p <- { pos with pos_fname = ""; pos_lnum = 1; } ;
     lex


```

./src/incLib.py

```
import sys
f = open(sys.argv[1])
s = f.readlines()
f.close()

#Get the include list
incLibs = []
codeLines = []
for i in s:
    j=i.strip()
    if len(j.split()) == 2 and j.split()[0]=="include":
        incLibs.append(j.split()[1])
        continue
    codeLines.append(i)

#Get all the library code to be attached
libCode = []
for lib in incLibs:
    f = open("library/"+lib+".plt")
    lc = f.readlines()
    f.close()
    libCode += lc

#Save the original code in .plt_tmp file
fn= open(sys.argv[1] + '_tmp','w')
for item in s:
    fn.write("%s" % item)
fn.close()

#overwrite the new with this code
fullCode = libCode + codeLines
fn= open(sys.argv[1],'w')
for item in fullCode:
    fn.write("%s" % item)
fn.close()


```

./src/Makefile

```
OBJS = ast.cmo err.cmo sast.cmo parser.cmo scanner.cmo semcheck.cmo codegen.cmo plotter.cmo

TARFILES = Makefile  scanner.mll parser.mly \
 ast.ml codegen.ml plotter.ml

plotter : $(OBJS)
 ocamlc -o plotter $(OBJS)

scanner.ml : scanner.mll
 ocamllex scanner.mll

parser.ml parser.mli : parser.mly
 ocamlyacc parser.mly

%.cmo : %.ml
 ocamlc -c $<

%.cmi : %.mli
 ocamlc -c $<

.PHONY : clean

clean :
 rm -f plotter parser.ml parser.mli scanner.ml *.svg testall.log \
 *.cmo *.cmi *.out *.diff
test:
 python test.py
#
ast.cmo:
ast.cmx:
sast.cmo:
sast.cmx:
semcheck.cmo:
sencheck.cmx:
codegen.cmo:  ast.cmo
codegen.cmx:  ast.cmx
plotter.cmo: scanner.cmo parser.cmi codegen.cmo \
    ast.cmo
plotter.cmx: scanner.cmx parser.cmx codegen.cmx \
    sast.cmx ast.cmx
parser.cmo: err.cmo ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: err.cmo parser.cmi
scanner.cmx: err.cmx parser.cmx
parser.cmi: ast.cmo
err.cmo:

```

./src/parser.mly

```
%{ open Ast
    open Lexing
    open Parsing

   let num_errors = ref 0

   let parse_error msg = (* called by parser function on error *)
   let start = symbol_start_pos() in
   let final = symbol_end_pos() in
   Printf.fprintf stdout "Characters: %d..%d: %s\n"
     (start.pos_cnum - start.pos_bol) (final.pos_cnum - final.pos_bol) msg;
         incr num_errors;
   flush stdout;
   exit 0
   %}

%token EOL LPAREN RPAREN LBRACK RBRACK
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN
%token EQUAL NEQ LESS GREATER LEQ GEQ
%token AND OR NOT
%token SEMI COMMA COMMENT OF COLON
%token STRING NUM BOOL POINT NONE LIST HASH
%token APPEND POP REMOVE AT LENGTH OF
%token TRUE FALSE
%token RETURN IF ELSE FOR WHILE END BREAK CONTINUE THEN FN
%token PRINTXY PRINT
%token LINE
%token <float> LIT_NUM
%token <string> LIT_STR
%token <string> ID
%token EOF


%nonassoc ELSE END BREAK CONTINUE

%right ASSIGN
%left AND OR
%left NOT
%left EQUAL NEQ
%left LESS GREATER LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD

%start program
%type < Ast.program> program

%%

program:
  stmt EOF { $1 }

stmt:
    /* nothing */       { { funcs=[]; main=[] } }
    | func_stmt stmt    { { funcs = $1::$2.funcs; main= $2.main} }
    | other_stmt stmt   { { funcs = $2.funcs; main= $1::$2.main} }
```

```
60  /* ================================================
61                        Variable
62     ================================================ */
63
64      literal:
65          | LIT_NUM       { Literal_Num($1) }
66          | LIT_STR       { Literal_Str($1) }
67          | point         { $1 }
68          | literal_list  { $1 }
69
70      literal_list:
71          | LBRACK list_elements RBRACK { Literal_List($2) }
72          | LBRACK list_elements { (parse_error "Syntax error: Left [ is
        unmatched with right ]."); }
73
74
75      list_elements:
76          /* nothing */ { [] }
77          | list_content  { List.rev $1 }
78
79      list_content:
80          expr                    { [$1] }
81          | list_content COMMA expr { $3 :: $1 }
82
83      point:
84          | LPAREN expr COMMA expr RPAREN     { Point($2, $4) }
85          | LPAREN expr COMMA expr { (parse_error "Syntax error: Left ( is
        unmatched with right )."); }
86          | LPAREN expr COMMA expr { (parse_error "Right ) is unmatched with
        left ( ."); }
87          | LPAREN expr expr RPAREN { (parse_error "Missing , ."); }
88          | LPAREN expr COMMA RPAREN { (parse_error "Missing y co-od of point
        ."); }
89          | LPAREN COMMA RPAREN { (parse_error "Missing x and y co-od of point
         ."); }
90          | LPAREN COMMA expr RPAREN { (parse_error "Missing x co-od of point
        ."); }
91
92
93      primitive:
94          | BOOL      {"bool"}
95          | NUM       {"num"}
96          | STRING    {"string"}
97          | POINT     {"point"}
98
99      /*non_primitive:
100         | LIST primitive */
101
102     data_type:
103         | primitive { $1 }
104         /*| non_primitive { $1 }*/
105         /* Point, List and hash are to be added here */
106
107     vdecl:
108         | vdecl_single      { $1 }
109
110     /* only to be used to declare one variable */
111     /* Reusability in functions */
112     vdecl_single:
113         | primitive_var_decl { $1 }
```

```
114         | list_decl { $1 }
115
116     primitive_var_decl:
117         | primitive ID  { Var_Decl($1, $2) }
118         | primitive { (parse_error "Missing variable name"); }
119
120     list_decl:
121         | LIST primitive ID { List_Decl($2, $3) }
122         | LIST ID           { (parse_error "Missing list type"); }
123         | LIST primitive    { (parse_error "Missing list name"); }
124
125
126     /*
127         For fututе to do, multiple variable declaration
128         eg : num a,b,c
129
130     id_list:
131         ID      {$1}
132         | ID COMMA id_list { $1, $3 }
133
134     */
135
136 /* ============================================
137                       Statements
138    ============================================ */
139
140
141     func_stmt:
142         | fdecl { $1 }
143
144     other_stmt:
145         | expr EOL            { Expr($1) }
146         | cond_stmt EOL       { $1 }
147         | list_stmt EOL       { $1 }
148         | assign_stmt EOL     { $1 }
149         | PRINTXY LPAREN expr COMMA expr RPAREN EOL     { PrintXY($3, $5) }
150         | PRINT expr EOL      { Print($2) }
151         | PRINT EOL           { (parse_error "Nothing to print!"); }
152         | line EOL            { $1 }
153         | fcall EOL           { $1 }
154         | RETURN expr EOL     { Return($2) }
155         | RETURN EOL           { (parse_error "Nothing to return!"); }
156         | vdecl EOL           { $1 }
157         | loop_stmt EOL       { $1 }
158         | EOL                 { Noexpr }
159
160     cond_stmt:
161         | IF expr COLON EOL other_stmt_list END            { Ifelse($2, $5,
    []) }
162         | IF expr EOL other_stmt_list END           { (parse_error "Missing
    colon :"); }
163         | IF expr COLON EOL other_stmt_list          { (parse_error "Missing
    end "); }
164         | IF COLON EOL other_stmt_list END           { (parse_error "Missing
    if condition "); }
165         | IF expr COLON EOL other_stmt_list ELSE COLON EOL other_stmt_list
    END  { Ifelse($2, $5, $9) }
166         | IF expr EOL other_stmt_list ELSE COLON EOL other_stmt_list END  {
    (parse_error "Missing colon : after if "); }
167         | IF expr COLON EOL other_stmt_list ELSE EOL other_stmt_list END  {
```

```
(parse_error "Missing colon : after else "); }
168         | IF expr EOL other_stmt_list ELSE COLON EOL other_stmt_list      {
(parse_error "Missing end "); }
169         | IF EOL other_stmt_list ELSE COLON EOL other_stmt_list END       {
(parse_error "Missing if condition"); }
170
171     list_stmt:
172         | ID OF APPEND LPAREN expr RPAREN       { Append( Id($1), $5)}
173         | ID OF POP LPAREN RPAREN               { Pop( Id($1) ) }
174         | ID OF REMOVE LPAREN expr RPAREN       { Remove( Id($1), $5 ) }
175
176     list_assign:
177         | ID ASSIGN literal_list {Assign(Id($1), $3) }
178         | ID literal_list        {(parse_error "Missing assignment operator
"); }
179         | ID ASSIGN              {(parse_error "Missing element(s) "); }
180
181     assign_stmt:
182         | ID ASSIGN expr        { Assign(Id($1), $3) }
183         | list_assign           { $1 }
184
185
186     line:
187         | LINE LPAREN ID COMMA ID RPAREN  { LineVar(Id($3), Id($5))}
188         | LINE LPAREN ID COMMA ID COMMA expr RPAREN  { LineVarColor(Id($3),
Id($5) , $7)}
189         | LINE  ID COMMA ID RPAREN  { (parse_error "Missing left paren ");}
190         | LINE LPAREN ID COMMA ID   { (parse_error "Missing right paren ");}
191         | LINE LPAREN ID ID RPAREN  { (parse_error "Missing , ");}
192         | LINE LPAREN LPAREN expr COMMA expr RPAREN COMMA LPAREN expr COMMA
expr RPAREN RPAREN { LineRaw($4, $6, $10, $12) }
193         | LINE LPAREN expr COMMA expr RPAREN COMMA LPAREN expr COMMA expr
RPAREN RPAREN { (parse_error "Missing left paren ");}
194         | LINE LPAREN LPAREN  expr COMMA expr RPAREN COMMA expr COMMA expr
RPAREN RPAREN { (parse_error "Missing left paren ");}
195         | LINE LPAREN LPAREN expr COMMA expr  COMMA LPAREN expr COMMA expr
RPAREN RPAREN { (parse_error "Missing right paren ");}
196         | LINE LPAREN LPAREN expr COMMA expr RPAREN COMMA LPAREN expr COMMA
expr RPAREN  { (parse_error "Missing right paren ");}
197         | LINE LPAREN LPAREN expr  expr RPAREN COMMA LPAREN expr COMMA expr
RPAREN RPAREN { (parse_error "Missing , ");}
198         | LINE LPAREN LPAREN expr COMMA expr RPAREN  LPAREN expr COMMA expr
RPAREN RPAREN { (parse_error "Missing , ");}
199         | LINE LPAREN LPAREN expr COMMA expr RPAREN COMMA LPAREN expr  expr
RPAREN RPAREN { (parse_error "Missing , ");}
200         | LINE LPAREN LPAREN expr COMMA expr RPAREN COMMA ID RPAREN { LinePX
($4, $6, Id($9)) }
201         | LINE LPAREN expr COMMA expr RPAREN COMMA ID RPAREN { (parse_error
"Missing left paren ");}
202         | LINE LPAREN LPAREN expr COMMA expr RPAREN COMMA ID { (parse_error
"Missing right paren ");}
203         | LINE LPAREN LPAREN expr COMMA expr COMMA ID RPAREN { (parse_error
"Missing right paren ");}
204         | LINE LPAREN LPAREN expr  expr RPAREN COMMA ID RPAREN { (parse_error
 "Missing , ");}
205         | LINE LPAREN LPAREN expr COMMA expr RPAREN ID RPAREN { (parse_error
 "Missing , ");}
206         | LINE LPAREN ID COMMA LPAREN expr COMMA expr RPAREN RPAREN { LinePX
($6, $8, Id($3)) }
207         | LINE  ID COMMA LPAREN expr COMMA expr RPAREN RPAREN { (parse_error
```

```
      "Missing left paren ");}
208       | LINE LPAREN ID COMMA  expr COMMA expr RPAREN RPAREN { (parse_error
      "Missing left paren ");}
209       | LINE LPAREN ID  LPAREN expr COMMA expr RPAREN RPAREN { (
      parse_error "Missing , ");}
210       | LINE LPAREN ID COMMA LPAREN expr  expr RPAREN RPAREN { (
      parse_error "Missing , ");}
211       | LINE LPAREN ID COMMA LPAREN expr COMMA expr  RPAREN { (parse_error
      "Missing right paren ");}
212
213     loop_stmt:
214       | FOR assign_stmt SEMI expr SEMI assign_stmt COLON EOL
      other_stmt_list END { For($2, $4, $6, List.rev $9) }
215       | FOR assign_stmt  expr SEMI assign_stmt COLON EOL other_stmt_list
      END { (parse_error "Missing ; ");}
216       | FOR assign_stmt SEMI expr  assign_stmt COLON EOL other_stmt_list
      END { (parse_error "Missing ; ");}
217       | FOR assign_stmt SEMI expr SEMI assign_stmt  EOL other_stmt_list
      END { (parse_error "Missing : ");}
218       | FOR assign_stmt SEMI expr SEMI assign_stmt COLON EOL
      other_stmt_list { (parse_error "Missing end ");}
219       | FOR  SEMI expr SEMI assign_stmt COLON EOL other_stmt_list END { (
      parse_error "Missing initialization statement ");}
220       | FOR assign_stmt SEMI  SEMI assign_stmt COLON EOL other_stmt_list
      END { (parse_error "Missing condition statement ");}
221       | FOR assign_stmt SEMI expr SEMI  COLON EOL other_stmt_list END { (
      parse_error "Missing increment/decrement statement ");}
222       | WHILE expr COLON EOL other_stmt_list END {While($2, List.rev $5)}
223       | WHILE expr COLON EOL other_stmt_list END { (parse_error "Missing :
      ");}
224       | WHILE expr COLON EOL other_stmt_list END { (parse_error "Missing
      end ");}
225       | WHILE expr COLON EOL other_stmt_list END { (parse_error "Missing
      expression in while ");}
226
227     other_stmt_list:
228       { [] }
229       | other_stmt_list other_stmt { $2 :: $1 }
230
231     stmt_list:
232        { [] }
233       | stmt_list other_stmt { $2 :: $1 }
234
235 /* ============================================
236                       Functions
237    ============================================ */
238
239    /* No locals. as variables can be declared at any point */
240
241     fdecl:
242       | FN ID LPAREN args_opt RPAREN COLON EOL stmt_list END EOL
243         { Fdecl({ fname = $2;
244           args = List.rev $4;
245           body = List.rev $8 }) }
246       | FN ID LPAREN args_opt RPAREN COLON EOL stmt_list END EOL { (
      parse_error "Missing colon : "); }
247       | FN ID args_opt RPAREN COLON EOL stmt_list END EOL        { (
      parse_error "Missing left paren "); }
248       | FN ID LPAREN args_opt COLON EOL stmt_list END EOL        { (
      parse_error "Missing right paren "); }
```

```
249         | FN LPAREN args_opt RPAREN EOL stmt_list END EOL          { (
      parse_error "Missing function name : "); }
250
251     args_opt:
252         { [] }
253         | args_list { List.rev $1 }
254
255     args_list:
256           arg                       { [$1] }
257         | args_list COMMA arg         { $3 :: $1 }
258         | args_list arg               { (parse_error "Missing , "); }
259         | args_list COMMA             { (parse_error "Missing arg "); }
260
261     arg:
262         vdecl_single                  { $1 }
263
264     /* Function Call */
265     fcall:
266         | ID LPAREN fparam RPAREN     { Fcall($1, $3) }
267
268     fparam:
269         {[]}
270         | expr                  { [$1] }
271         | fparam COMMA expr      { $3 :: $1 }
272         | fparam  expr           { (parse_error "Missing , "); }
273         | fparam COMMA           { (parse_error "Missing expr "); }
274
275
276 /* ============================================
277                     Expressions
278    ============================================ */
279
280   expr:
281   | arith_expr           { $1 }
282   | log_expr             { $1 }
283   | LPAREN expr RPAREN   { $2 }
284   | LPAREN  RPAREN       { (parse_error "Missing expression "); }
285   | LPAREN expr          { (parse_error "Missing right paren "); }
286
287    log_expr:
288   | expr EQUAL  expr        { Binop($1, Equal, $3) }
289   | expr NEQ  expr          { Binop($1, Neq,   $3) }
290   | expr LESS  expr         { Binop($1, Less,  $3) }
291   | expr LEQ  expr          { Binop($1, Leq,   $3) }
292   | expr GREATER  expr      { Binop($1, Greater,  $3) }
293   | expr GEQ  expr          { Binop($1, Geq,   $3) }
294   | expr EQUAL              { (parse_error "Missing second  expression "); }
295   | expr NEQ                { (parse_error "Missing second  expression "); }
296   | expr LESS               { (parse_error "Missing second  expression "); }
297   | expr LEQ                { (parse_error "Missing second  expression "); }
298   | expr GREATER            { (parse_error "Missing second  expression "); }
299   | expr GEQ                { (parse_error "Missing second  expression "); }
300   | log_expr AND log_expr   { Binop($1, And, $3) }
301   | log_expr OR log_expr    { Binop($1, Or, $3) }
302   | log_expr OR             { (parse_error "Missing second  expression "); }
303   | log_expr AND            { (parse_error "Missing second  expression "); }
304
305
306   arith_expr :
307   | LPAREN arith_expr RPAREN     { $2 }
```

```
308     | LPAREN arith_expr            { (parse_error "Missing right paren "); }
309     | LPAREN RPAREN                { (parse_error "Missing expr "); }
310     | arith_expr PLUS   arith_expr { Binop($1, Add,   $3) }
311     | arith_expr MINUS  arith_expr { Binop($1, Sub,   $3) }
312     | arith_expr TIMES  arith_expr { Binop($1, Mul,  $3) }
313     | arith_expr DIVIDE arith_expr { Binop($1, Div,   $3) }
314     | arith_expr MOD arith_expr    { Binop($1, Mod,   $3) }
315     | arith_expr PLUS              { (parse_error "Missing second arithmetic
        expression "); }
316     | arith_expr MINUS             { (parse_error "Missing second arithmetic
        expression "); }
317     | arith_expr TIMES             { (parse_error "Missing second arithmetic
        expression "); }
318     | arith_expr DIVIDE            { (parse_error "Missing second arithmetic
        expression "); }
319     | arith_expr MOD               { (parse_error "Missing second arithmetic
        expression "); }
320     | list_exprs                 { $1 }
321     | atom               { $1 }
322
323     list_exprs:
324     | ID OF LENGTH LPAREN  RPAREN          { Length( Id($1) ) }
325     | ID LENGTH LPAREN  RPAREN             { (parse_error "Missing . "); }
326     | ID OF LENGTH  RPAREN                 { (parse_error "Missing left paren
          "); }
327     | ID OF LENGTH LPAREN                  { (parse_error "Missing right
        paren "); }
328     | ID OF AT LPAREN expr RPAREN          { Access( Id($1), $5 ) }
329     | ID  AT LPAREN expr RPAREN            { (parse_error "Missing of "); }
330     | ID OF AT expr RPAREN                 { (parse_error "Missing left paren
          "); }
331     | ID OF AT LPAREN expr                 { (parse_error "Missing right
        paren "); }
332     | ID LBRACK expr RBRACK                { Access( Id($1), $3 ) }
333     | ID LBRACK RBRACK                     { (parse_error "Missing expr "); }
334     | ID LBRACK expr                       { (parse_error "Missing right
        brack "); }
335
336
337
338     atom:
339     | literal          { $1 }
340     | TRUE             { Bool(True) }
341     | FALSE            { Bool(False) }
342     | ID               { Id($1) }
343
344
```

./src/plotter.ml

```
1
2  type action = Ast | Codegen (* | Tast | Cppsast *)
3
4  let _ =
5    let action = if Array.length Sys.argv > 1 then
6      List.assoc Sys.argv.(1) [ ("-a", Ast);
7                  ("-c", Codegen); (* ("-t", Tast) *) ]
8    else Codegen in
9    let lexbuf = Lexing.from_channel stdin in
10   let program = Parser.program Scanner.token lexbuf in
11   match action with
12     Ast -> print_string (Ast.string_of_program (program))
13   | Codegen -> print_string (Codegen.convert (program))
14   (*| Tast -> Cppsast.convert_to_cppast (List.rev program)*)
15
16
```

./src/plt

```
#!/bin/bash

# Simple runnable shell script to avoid the manual work
# This script runs the plotter code, gets the c++ outut
# compiles  it  and executes its output(which generates
# the svg). This is then renamed, & temp files are
# removed

# 1st arg - input .plt file

f=$1
l=${#f}

file=${f::l-4}

#attach the libraries
python incLib.py $1

#EOL EOF issue fix
printf "\n" >> $1

./plotter < $1 > $file.cpp

g++ -std=c++11 $file.cpp

./a.out

mv hello.svg $file.svg

#reset the file to normal
rm $file.plt
cp $file.plt_tmp $file.plt
rm $file.plt_tmp

#remove the cpp and out files
rm $file.cpp a.out


```

./src/sast.ml

```
1
2  open Ast
3
4  type t =  Num | Bool | String | Point | List
5          | ListNum | ListString | ListPoint | ListBool
6
7  type texpr =
8        Literal_Num of float * t
9      | Literal_Str of string * t
10     | Literal_List of texpr list * t
11     | Point of texpr * texpr * t
12     | Binop of texpr * Ast.ops * texpr * t
13     | Id of string * t
14     | Bool of bool * t
15     | Length  of texpr * t
16     | Access of texpr * texpr * t * t
17
18
19 type tstmt =
20     Expr of texpr * t
21   | Var_Decl of string * string * t
22   | List_Decl of string * string *  t
23   | Passign of texpr * texpr
24   | Assign of texpr * texpr
25   | Append of texpr * texpr
26   | Remove of texpr * texpr
27   | Pop    of texpr
28   | Fcall  of string * texpr list
29   | PrintXY of texpr * texpr
30   | Print of texpr
31   | LineVar of texpr * texpr
32   | LineVarColor of texpr * texpr * texpr
33   | LineRaw of texpr * texpr * texpr * texpr
34   | LinePX of texpr * texpr * texpr
35   | For of tstmt * texpr * tstmt * tstmt list
36   | While of texpr * tstmt list
37   | Ifelse of texpr * tstmt list * tstmt list
38   | Return of texpr
39   | Noexpr
40   | Fdecl of fdecl and
41       fdecl = {
42        fname : string;
43        args  : tstmt list;
44        body  : tstmt list;
45      }
46
47 type program = tstmt list
48
49 (* Pretty Print Stuff *)
50
51 let typeof t =
52   match t with
53         | Num -> "num"
54         | Bool -> "bool"
55         | String -> "string"
56         | Point -> "point"
57         | ListNum    -> "listNum"
58         | ListString -> "listString"
59         | ListBool   -> "listBool"
```

```
60          | ListPoint  -> "listPoint"
61          | List -> "list"
62
63
64  let rec string_of_texpr = function
65      Literal_Num(l, t) -> string_of_float l ^ typeof t
66    | Literal_Str(l, t) -> l ^ typeof t
67    | Point(e1, e2, t) -> "(" ^ string_of_texpr e2 ^ "," ^ string_of_texpr e2
      ^ ")" ^ typeof t
68    | Literal_List(l, t) ->  typeof t
69    | Id(s, t) -> s ^ typeof t
70    | Length(v,_)  -> string_of_texpr v ^ ".length()"
71    | Binop(e1, o, e2, t) ->
72        string_of_texpr e1 ^ " " ^
73        (match o with
74    Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/"
75        | Equal -> "==" | Neq -> "!="
76        | Mod -> "%"
77        | And -> "&&" | Or ->"||"
78        | Square -> "**"
79        | Less -> "<" | Leq -> "<="
80        | Greater -> ">" | Geq -> ">="
81        ) ^ " " ^ string_of_texpr e2 ^ typeof t
82    | Bool(x, t) -> if x = True then "true" else "false" ^ typeof t
83    | Access(v, e, t1, t2) -> string_of_texpr v ^ ".at(" ^ ( string_of_texpr e
      ) ^ ")" ^ " //of type " ^ typeof t1 ^ typeof t2
84
85
86
87  let rec string_of_tstmt = function
88      Expr(expr, t) -> string_of_texpr expr ^ "\n" ^ typeof t
89    | Noexpr -> ""
90    | Var_Decl(tp, id, t) -> tp ^ " " ^ id ^ "\n" ^ typeof t
91    | List_Decl(tp, id, t) -> "list" ^ tp ^ " " ^ id ^ "\n" ^ typeof t
92    | Passign(v, e1) -> string_of_texpr v ^ " = " ^ ( string_of_texpr e1 ) ^ "
      \n"
93    | Assign(v, e) -> string_of_texpr v ^ " = " ^ ( string_of_texpr e )
94    | Append(v, e) -> string_of_texpr v ^ ".append(" ^ ( string_of_texpr e ) ^
      ")"
95    | Remove(v, e) -> string_of_texpr v ^ ".remove(" ^ ( string_of_texpr e ) ^
      ")"
96    | Pop(v) -> string_of_texpr v ^ ".pop()"
97    | Fcall(v, el) ->  v ^ "("^ (String.concat "," (List.map string_of_texpr
      el)) ^")\n"
98    | PrintXY(e1,e2) -> "printXY(" ^ string_of_texpr e1 ^ "," ^
      string_of_texpr e2 ^ ")\n"
99    | Print(e) -> "print " ^ string_of_texpr e ^ "\n"
100   | LineVar(e1,e2)-> "line (" ^ string_of_texpr e1 ^ "," ^ string_of_texpr
      e2 ^ ")" ^ "\n"
101   | LineVarColor(e1,e2, c)-> "line (" ^ string_of_texpr e1 ^ "," ^
      string_of_texpr e2 ^ "," ^ string_of_texpr c ^")" ^ "\n"
102   | LineRaw(e1,e2,e3,e4)-> "line ( (" ^ string_of_texpr e1 ^ "," ^
      string_of_texpr e2 ^ ")" ^ "," ^ "("
103                           ^ string_of_texpr e3 ^ "," ^ string_of_texpr e4
      ^ ") )\n"
104   | LinePX(e1, e2, e3)-> "line ( ( " ^ string_of_texpr e1 ^ "," ^
      string_of_texpr e2 ^ ") ," ^ string_of_texpr e3 ^ ") \n"
105   | For(s1, e1, s2, body) -> "for " ^ string_of_tstmt s1 ^ " ; " ^
      string_of_texpr e1 ^ " ; " ^ string_of_tstmt s2 ^ ": \n"
106                           ^ ( String.concat "\n\t" (List.map
```

```
     string_of_tstmt body) )
107                                  ^ "\nend\n"
108     | While(e, body) -> "while " ^ string_of_texpr e ^ " :\n" ^ (String.concat
         "\n\t" (List.map string_of_tstmt body)) ^ "\nend\n"
109     | Ifelse(e, s1, s2) -> "if " ^ string_of_texpr e ^ " :\n" ^ (String.concat
         "\n\t" (List.map string_of_tstmt s1)) ^ "\nelse:\n" ^ (String.concat "\n\
        t" (List.map string_of_tstmt s2)) ^ "\nend\n"
110     | Return(expr) -> "return " ^ string_of_texpr expr ^ "\n"
111     | Fdecl(f) -> string_of_fdecl f and
112     string_of_fdecl fdecl =
113         "fn " ^ fdecl.fname ^ "(" ^
114           ( String.concat ", " (List.map (fun s -> string_of_tstmt s ) fdecl.
        args) ) ^
115           "):\n" ^
116         ( String.concat "" (List.map string_of_tstmt fdecl.body) ) ^
117         "\nend\n"
118   let string_of_tprogram stmts =
119     String.concat "\n" (List.map string_of_tstmt stmts)
120
121
```

./src/scanner.mll

```
{
  open Err
  open Parser
  open Lexing
}

rule token = parse
  [' ' '\t' '\r'] { token lexbuf } (*whitespace*)
(*punctuations*)
| '\n' { EOL }
| "*/" { token lexbuf }
| '+' { PLUS } (*operators*)
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '%' { MOD }
| ">=" { GEQ }
| "<=" { LEQ }
| "==" { EQUAL }
| "!=" { NEQ }
(*| "**" { SQUARE }*)
| '>' { GREATER }
| '<' { LESS }
| '(' { LPAREN }
| ')' { RPAREN }
| '[' { LBRACK }
| ']' { RBRACK }
| ',' { COMMA }
| '.' { OF }
| '=' { ASSIGN }
| "and" { AND }
| "or" { OR }
(*| "in" { IN }*)
| "not" { NOT }
| ';' { SEMI }
| ':' { COLON }
| "string" { STRING }
| "num" { NUM }
| "bool" { BOOL }
| "point" { POINT }
| "if" { IF } (*controlling sequence*)
| "else" { ELSE }
| "then" { THEN }
| "end" { END }
| "for" { FOR }
| "while" { WHILE }
| "break" { BREAK }
| "continue" { CONTINUE }
| "print" { PRINT }
| "printXY" { PRINTXY }
| "line" { LINE }
| "none" { NONE }
| "list" { LIST }
| "hash" { HASH }

(* List related stuff *)
(* the '.' in front of append ensures its used as a function *)
| "append" { APPEND }
| "pop"    { POP }
```

```
60  | "remove" { REMOVE }
61  | "at"     { AT }
62  | "length" { LENGTH }
63
64  | "fn" { FN }
65  | "return" { RETURN }
66  | "true" { TRUE }
67  | "false"  { FALSE }
68  | '-'?(['0'-'9']+('.'['0'-'9']*)?) as lxm { LIT_NUM(float_of_string lxm) }
      (*Change to add negative*)
69  | ['"'][^'"']*['"'] as str { LIT_STR(str) }
70  | ['A'-'Z' 'a'-'z']+['A'-'Z' 'a'-'z' '0'-'9']* as i { ID(i) }
71  | eof { EOF }
72  | '#'    { singleLineComment lexbuf }
73  | "/*"   { multiLineComment lexbuf }
74  | _ as c { raise (Failure("Illegal character : " ^ Char.escaped c)) }
75
76  and singleLineComment = parse
77  | '\n'  {token lexbuf}
78  | _     {singleLineComment lexbuf}
79
80  and multiLineComment = parse
81  |"*/"    { token lexbuf }
82  | _      { multiLineComment lexbuf }
83
```

./src/semcheck.ml

```
1  open Sast
2  open Ast
3
4  module StringMap = Map.Make(String)
5
6  type s_env =
7          {       var_types: string StringMap.t ref list;
8                  var_inds : int StringMap.t ref list;
9                  f_list   : string StringMap.t ref list;
10         }
11
12 let check stmts =
13
14     let fail msg = (* raise (Failure msg) *)
15                    print_string "Error : ";
16                    print_string msg;
17                    print_string "\n";
18                    exit 0
19     in
20
21     let find_max_index map =
22         let bindings = StringMap.bindings map in
23         let rec max cur = function
24             | [] -> cur
25             | hd :: tl -> if snd hd > cur then max (snd hd) tl else max cur
    tl
26         in
27         max 0 bindings
28     in
29
30     let type_to_str t = match t with
31         | Sast.Num          -> "num"
32         | Sast.String       -> "string"
33         | Sast.Bool         -> "bool"
34         | Sast.Point        -> "point"
35         | Sast.ListBool     -> "listbool"
36         | Sast.ListNum      -> "listnum"
37         | Sast.ListString   -> "liststring"
38         | Sast.ListPoint    -> "listpoint"
39         | Sast.List         -> "list"
40
41     in
42
43     (* Setting Environment for Sast *)
44     let find_var var map_list =
45         let rec finder var = function
46             | m :: tl ->
47                 (try (StringMap.find var !m)
48                  with
49                  | Not_found -> finder var tl)
50             | [] -> raise (Not_found )
51         in
52         finder var map_list
53     in
54
55     (* build default symbol tables: *)
56     let sast_env =
57         {       var_types    = [ref StringMap.empty];
58                 var_inds     = [ref StringMap.empty];
```

```
59                  f_list      =  [ref StringMap.empty];
60          }
61
62      in
63
64      (*Check parts*)
65
66
67
68      let typeof elem = match elem with
69          | Sast.Literal_Num(_,t) -> t
70          | Sast.Literal_Str(_,t) -> t
71          | Sast.Point(_,_,t) -> t
72          | Sast.Literal_List(_,t) -> t
73          | Sast.Binop(_,_,_,t) -> t
74          | Sast.Id(_,t) -> t
75          | Sast.Bool(_,t) -> t
76          | Sast.Length(_,t) -> t
77          | Sast.Access(_,_,_,t) -> t
78      in
79
80      (* COnverting Ast to Sast *)
81      (* uses the sast_env and statment list which is recieved as input *)
82      let convert_to_sast stmts_list env=
83
84          let rec expr env = function
85              | Ast.Literal_Num(v) -> Sast.Literal_Num(v, Sast.Num)
86              | Ast.Literal_Str(v) -> Sast.Literal_Str(v, Sast.String)
87              | Ast.Point(e1, e2) ->
88                  let se1 = expr env e1 in
89                  let se2 = expr env e2 in
90                  let te1 = typeof se1 in
91                  let te2 = typeof se2 in
92                  if( te1=te2 && te1=Sast.Num)
93                  then Sast.Point(se1, se2, Sast.Point)
94                  else fail("Point's value should only be of type num.")
95              | Ast.Literal_List(v) ->
96                      let tv = List.map (fun s -> expr env s) v in
97                      (match tv with
98                      | [] -> Sast.Literal_List(tv, Sast.List)
99                      | hd::tl ->
100                         (match (type_to_str (typeof hd)) with
101                         | "num"     -> Sast.Literal_List(tv, Sast.ListNum)
102                         | "string"  -> Sast.Literal_List(tv, Sast.ListString
    )
103                         | "bool"    -> Sast.Literal_List(tv, Sast.ListBool)
104                         | _         -> fail("Lists can contain only bool/
    string/num")
105                         )
106                     )
107
108             | Ast.Bool(v)        -> Sast.Bool(v,Sast.Bool)
109             | Ast.Id(v)          ->
110                 (* uses find_var to determine the type of id *)
111                 (try
112                     let tp = find_var v env.var_types in
113                     ( match tp with
114                         | "num"         ->  Sast.Id(v, Sast.Num)
115                         | "string"      ->  Sast.Id(v,Sast.String)
116                         | "point"       ->  Sast.Id(v,Sast.Point)
```

```
117                         | "bool"        ->  Sast.Id(v, Sast.Bool)
118                         | "listnum"     ->  Sast.Id(v, Sast.ListNum)
119                         | "liststring"  ->  Sast.Id(v, Sast.ListString)
120                         | "listpoint"   ->  Sast.Id(v, Sast.ListPoint)
121                         | "listbool"    ->  Sast.Id(v, Sast.ListBool)
122                         | _             ->  fail(" Invalid syntax..")
123                     )
124                  with
125                   | Not_found -> fail ("undeclared variable: " ^ v)
126                 )
127             | Ast.Access(v, e) ->
128                 let sv = expr env v in
129                 let se = expr env e in
130                 let tv = typeof sv in
131                 let te = typeof se in
132                 if ( te=Sast.Num )
133                 then (
134                     if (tv = Sast.ListNum)
135                     then Sast.Access(sv, se, Sast.ListNum, Sast.Num)
136                     else (
137                         if (tv= Sast.ListString)
138                         then Sast.Access(sv, se, Sast.ListString, Sast.
    String)
139                         else (
140                             if (tv= Sast.ListBool)
141                             then Sast.Access(sv, se, Sast.ListBool, Sast.
    Bool)
142                             else (
143                                 if (tv= Sast.ListPoint)
144                                 then Sast.Access(sv, se, Sast.ListPoint,
    Sast.Point)
145                                 else (if (tv= Sast.Point)
146                                     then Sast.Access(sv, se, Sast.Point,
    Sast.Num)
147                                     else ( fail("'access' operations can be
    performed only on List variables. Here its applied on "^ type_to_str tv)
148                                         )
149                                 )
150                             )
151                         )
152                     )
153                 )
154                 else fail ("The 'index' in list_elem.at(index)  should be of
     num type only." ^ (type_to_str tv))
155             | Ast.Length(v) ->
156                 let sv = expr env v in
157                 let tv = typeof sv in
158                 if ( (tv = Sast.ListNum) || (tv = Sast.ListPoint) || (tv =
    Sast.ListString) || (tv = Sast.ListBool))
159                 then Sast.Length(sv, Sast.Num)
160                 else fail ("'length()' can be performed only on List
    variables.")
161             | Ast.Binop(e1, op, e2) ->
162                 let se1 = expr env e1 in
163                 let se2 = expr env e2 in
164                 let e1_data = typeof se1 in
165                 let e2_data = typeof se2 in
166                 (match op with
167                 | Add | Sub | Mul | Div ->
168                     (match e1_data with
```

```
169                     | Num ->
170                         (match e2_data with
171                             | Num -> Sast.Binop(se1, op, se2, Sast.Num)
172                             | String ->  fail ("Cannot Add Num and string")
173                             | Bool -> fail ("Cannot Add Bool")
174                             | _ -> fail ("Incorrect type " ^ (type_to_str
    e2_data) ^ " with Num"))
175                     | _ -> fail ("Operation on incompatible types")
176                     )
177                 | Equal | Neq  ->
178                     (match e1_data with
179                     | Num ->
180                         (match e2_data with
181                             | Num -> Sast.Binop(se1, op, se2, Sast.Bool)
182                             | _ -> fail("Incorrect type with Num == or != ")
183                         )
184                     | String ->
185                          (match e2_data with
186                             | String -> Sast.Binop(se1, op, se2, Sast.Bool)
187                             | _ -> fail("Incorrect type with String == or !=
    ")
188                         )
189                     | Bool ->
190                          (match e2_data with
191                             | Bool -> Sast.Binop(se1, op, se2, Sast.Bool)
192                             | _ -> fail("Incorrect type with Bool == or != "
    )
193                         )
194                     | _  -> fail("Type which is not num, string or bool
    cannot be used in equal or neq")
195                     (*| Void  -> fail ("Cannot perform binop on void") *)
196                     )
197
198                 | And | Or  ->
199                     if ( (e1_data=Num || e1_data=Bool ) && (e2_data=Num ||
    e2_data=Bool) )
200                     then Sast.Binop(se1, op, se2, Sast.Bool)
201                     else fail("Incorrect type with 'and' and 'or'")
202
203                 | Less | Leq | Greater | Geq ->
204                     (match e1_data with
205                     | Num ->
206                         (match e2_data with
207                             | Num -> Sast.Binop(se1, op, se2, Sast.Bool)
208                             | _ -> fail("Incorrect type with Num < or <= or
    > or >= ")
209                         )
210                     | String ->
211                          (match e2_data with
212                             | String -> Sast.Binop(se1, op, se2, Sast.Bool)
213                             | _ -> fail ("cannot mix string and  < or <= or
    > or >= ")
214                         )
215                     | _ -> fail ("Cannot perform less and grt ops on these
    types")
216                     )
217                 | Mod | Square ->
218                     (match e1_data with
219                     | Num ->
220                         (match e2_data with
```

```
221                             | Num -> Sast.Binop(se1, op, se2, Sast.Bool)
222                             | _ -> fail("Incorrect type for mod. both should
    be num")
223                         )
224                     | _ -> fail("Mod & square can only be used with num")
225                     )
226                 )
227         in
228
229         (* Convert ast stmt to sast stmt *)
230         let rec stmt env = function
231             | Ast.Noexpr -> Sast.Noexpr
232             | Ast.Expr(e) ->
233                 let se = expr env e in
234                 let tp = typeof se in
235                 Sast.Expr(se, tp)
236             | Ast.Passign(v,e1,e) ->
237                 let sv = expr env v in
238                 let se1 = expr env e1 in
239                 let tv = typeof sv in
240                 let te1 = typeof se1 in
241                 if ( tv = Sast.Point && te1 = tv )
242                     then Sast.Passign(sv, se1)
243                 else fail ("Invalid type assign. cannot assign " ^ (
    type_to_str te1) ^ " to type " ^ (type_to_str tv))
244             | Ast.Assign(v,e) ->
245                 let sv = expr env v in
246                 let se = expr env e in
247                 let tv = typeof sv in
248                 let te = typeof se in
249                 if ( tv = te || (te=Sast.List && (tv=Sast.ListNum || tv=Sast
    .ListString || tv=Sast.ListBool || tv=Sast.ListPoint) ) )
250                 then Sast.Assign(sv, se)
251                 else fail ("Invalid type assign. cannot assign " ^ (
    type_to_str te) ^ " to " ^ (type_to_str tv))
252             | Ast.Append(v, e) ->
253                 let sv = expr env v in
254                 let se = expr env e in
255                 let tv = typeof sv in
256                 let te = typeof se in
257                 if ( (tv = Sast.ListNum && te=Sast.Num) || (tv = Sast.
    ListPoint && te=Sast.Point ) || (tv = Sast.ListString && te=Sast.String)
    || (tv = Sast.ListBool && te=Sast.Bool))
258                 then Sast.Append(sv, se)
259                 else fail ("Invalid type append. cannot append " ^ (
    type_to_str te) ^ " to " ^ (type_to_str tv))
260             | Ast.Remove(v, e) ->
261                 let sv = expr env v in
262                 let se = expr env e in
263                 let tv = typeof sv in
264                 let te = typeof se in
265                 if ( (tv = Sast.ListNum) || (tv = Sast.ListPoint) || (tv =
    Sast.ListString) || (tv = Sast.ListBool))
266                 then
267                     if ( te=Sast.Num )
268                     then Sast.Remove(sv, se)
269                     else fail("The 'index' in *.pop(index) should be of num
    type only. It cannot be of type " ^ (type_to_str te))
270                 else fail ("'access' operations can be performed only on
    List variables.")
```

```
271             | Ast.Pop(v) ->
272                 let sv = expr env v in
273                 let tv = typeof sv in
274                 if ( (tv = Sast.ListNum) || (tv = Sast.ListPoint) || (tv =
    Sast.ListString) || (tv = Sast.ListBool))
275                 then Sast.Pop(sv)
276                 else fail ("'pop()' can be performed only on List variables.
    ")
277             | Ast.Fcall(v, el) ->
278                 let sel = List.map (fun s -> expr env s) el in
279                 (* Check if function is present *)
280                 Sast.Fcall(v, sel)
281             | Ast.Var_Decl(dt, id) ->
282                 (try
283                 ignore (StringMap.find id !(List.hd env.var_types));
284                     fail ("variable already declared in local scope: " ^ id)
285                  with | Not_found -> (List.hd env.var_types) := StringMap.
    add id dt !(List.hd env.var_types);
286                             (List.hd env.var_inds) := StringMap.add id (
    find_max_index !(List.hd env.var_inds)+1) !(List.hd env.var_inds);
287                       | Failure(f) -> raise (Failure (f) )
288                 );
289                 let tp = find_var id env.var_types in
290                 if (tp="num") then
291                         Sast.Var_Decl(dt, id, Sast.Num)
292                     else
293                         if (tp="string") then
294                             Sast.Var_Decl(dt, id, Sast.String)
295                         else
296                             if (tp="point") then
297                                 Sast.Var_Decl(dt, id, Sast.Point)
298                             else
299                                 Sast.Var_Decl(dt, id, Sast.Bool)
300             | Ast.List_Decl(dt, id) ->
301                 (try
302                 ignore (StringMap.find id !(List.hd env.var_types));
303                     fail ("variable already declared in local scope: " ^ id)
304                  with | Not_found -> (List.hd env.var_types) := StringMap.
    add id ("list"^dt) !(List.hd env.var_types);
305                             (List.hd env.var_inds) := StringMap.add id (
    find_max_index !(List.hd env.var_inds)+1) !(List.hd env.var_inds);
306                       | Failure(f) -> raise (Failure (f) )
307                 );
308                 let tp = find_var id env.var_types in
309                 if (tp="listnum") then
310                     Sast.List_Decl(dt, id, Sast.ListNum)
311                 else
312                     if (tp="liststring") then
313                         Sast.List_Decl(dt, id, Sast.ListString)
314                     else
315                         if (tp="listpoint") then
316                             Sast.List_Decl(dt, id, Sast.ListPoint)
317                         else
318                             Sast.List_Decl(dt, id, Sast.ListBool)
319             | Ast.Print(e) ->
320                 let se1 = expr env e in
321                 let te1 = typeof se1 in
322                 Sast.Print(se1)
323             | Ast.PrintXY(e1,e2) ->
324                 let se1 = expr env e1 in
```

```
325                 let se2 = expr env e2 in
326                 let te1 = typeof se1 in
327                 let te2 = typeof se2 in
328                 if (te2 = Sast.Point)
329                 then Sast.PrintXY(se1, se2)
330                 else fail("The second argument of printXY should be of type
    point")
331             | Ast.LineVar(e1, e2) ->
332                 let se1 = expr env e1 in
333                 let se2 = expr env e2 in
334                 let te1 = typeof se1 in
335                 let te2 = typeof se2 in
336                 if( te1 = Sast.Point && te2 = Sast.Point )
337                 then Sast.LineVar(se1, se2)
338                 else fail ("LineVar has to be called with 2 points")
339             | Ast.LineVarColor(e1, e2,c) ->
340                 let se1 = expr env e1 in
341                 let se2 = expr env e2 in
342                 let sc  = expr env c in
343                 let te1 = typeof se1 in
344                 let te2 = typeof se2 in
345                 let tc = typeof sc in
346                 if( te1 = Sast.Point && te2 = Sast.Point && tc=Sast.String )
347                 then Sast.LineVarColor(se1, se2, sc)
348                 else fail ("Line has to be called with 2 points and string
    for color (optional)")
349             | Ast.LineRaw(e1, e2, e3, e4) ->
350                 let se1 = expr env e1 in
351                 let se2 = expr env e2 in
352                 let se3 = expr env e3 in
353                 let se4 = expr env e4 in
354                 let te1 = typeof se1 in
355                 let te2 = typeof se2 in
356                 let te3 = typeof se3 in
357                 let te4 = typeof se4 in
358                 if( te1 = Sast.Num && te2 = Sast.Num && te3 = Sast.Num &&
    te4 = Sast.Num )
359                 then Sast.LineRaw(se1, se2, se3, se4)
360                 else fail ("LineRaw has to be called with 4 nums")
361             | Ast.LinePX(e1, e2, e3) ->
362                 let se1 = expr env e1 in
363                 let se2 = expr env e2 in
364                 let se3 = expr env e3 in
365                 let te1 = typeof se1 in
366                 let te2 = typeof se2 in
367                 let te3 = typeof se3 in
368                 if( te1 = Sast.Num && te2 = Sast.Num && te3 = Sast.Point )
369                 then Sast.LinePX(se1, se2, se3)
370                 else fail ("Line has to be called with 2 points")
371             | Ast.For(s1, e1, s2, body) ->
372                 let ss1 = stmt env s1 in
373                 let se1 = expr env e1 in
374                 let ss2 = stmt env s2 in
375                 Sast.For(ss1, se1, ss2, List.map (fun s -> stmt env s) body)
376             | Ast.While(e, body) ->
377                 let se = expr env e in
378                 let te = typeof se in
379                 if ( te = Sast.Num || te = Sast.Bool)
380                 then Sast.While(se, List.map (fun s -> stmt env s) body)
381                 else fail("The condition in while should give eiether num or
```

```
 bool. Not of type " ^type_to_str te)
382             | Ast.Ifelse(e, s1, s2) ->
383                 let se = expr env e in
384                 Sast.Ifelse(se, List.map (fun s -> stmt env s) s1, List.map
    (fun s -> stmt env s) s2)
385             | Ast.Return(e) -> Sast.Return(expr env e)
386             | Ast.Fdecl(f) ->
387                     let fnEnv = {
388                             var_types   =  [ref StringMap.empty];
389                             var_inds    =  [ref StringMap.empty];
390                             f_list      =  [ref StringMap.empty];
391                     } in
392                     let f_name =
393                         (try
394                         ignore (StringMap.find f.fname !(List.hd env.f_list)
    );
395                             fail ("Function already declared in local scope:
     " ^ f.fname)
396                          with | Not_found -> (List.hd env.f_list) :=
    StringMap.add f.fname "function" !(List.hd env.f_list);
397                               | Failure(f) -> raise (Failure (f) )
398                         );
399
400
401                     in
402                     let fargs = List.map (fun s -> stmt fnEnv s) f.args in
403                     let fstms = List.map (fun s -> stmt fnEnv s) f.body in
404                     Sast.Fdecl({
405                         fname = f.fname;
406                         args  = fargs;
407                         body  = fstms;
408                     })
409         in
410         List.map (fun s -> stmt env s) stmts_list
411      in
412      Sast.string_of_tprogram (convert_to_sast stmts sast_env)
413
```

./src/tast.ml

```
(* operations *)
type  ops =
        | Add | Sub | Mul  | Div | Mod
        | Equal | Neq | Less | Leq | Greater | Geq
        | And | Or
        | Square

type bool =
        | True | False


(*expressions*)
type texpr =
    Literal_Num of float
  | Literal_Str of string
  | Point of texpr * texpr
  | Literal_List of texpr list                (* Eg [ texpr, texpr, .. ] *)
  | Binop of texpr * ops * texpr               (* Binary Ops *)
  | Id of string                              (* identifiers *)
  | Bool of bool                              (* True *)
  | Length of texpr                            (* a.length() *)
  | Access of texpr * texpr                    (* a.at(3), a[3] *)


type tstmt = (* Statements *)
    Texpr of texpr
  | Var_Decl of string * string               (* (type, id) *)
  | List_Decl of string * string
  | Passign of texpr * texpr * tstmt                    (* (type, p1, p2) *)
  | Assign of texpr * texpr                    (* a = 2 *)
  | Append of texpr * texpr                    (* a.append(7) *)
  | Pop of texpr                              (* a.pop() *)
  | Remove of texpr * texpr                    (* a.remove(3) *)
  | Fcall  of string * texpr list              (* a.() *)
  | PrintXY of texpr * texpr                   (* printXY ( 5 , (1,2)) *)
  | Print of texpr                            (* print 5 *)
  | LineVar of texpr * texpr                   (* line(p,q) *)
  | LineRaw of texpr * texpr * texpr * texpr     (* line((3,4), (7,9)) *)
  | LinePX of texpr * texpr * texpr             (* line((3,4), x) , line(x,
   (3,4)) *)
  | For of tstmt * texpr * tstmt * tstmt list    (* for i=0; i<5; i=i+1: *)
  | While  of texpr * tstmt list
  | Ifelse of texpr * tstmt list * tstmt list
  | Return of texpr
  | Noexpr
  | Fdecl of fdecl and
      fdecl = {
       fname : string;
       args  : tstmt list;
       body  : tstmt list;
      }


type tprogram = {
                    funcs : tstmt list;
                    main  : tstmt list;
               }
```

```
59
60
61
62  (* Pretty Print
63
64  let rec string_of_texpr = function
65      Literal_Num(l) -> string_of_float l ^ "0"
66    | Literal_Str(l) -> l
67    | Point(e1, e2) -> "(" ^ string_of_texpr e1 ^ "," ^ string_of_texpr e2 ^ "
      )"
68    | Literal_List(l) -> "[" ^ (String.concat "," (List.map string_of_texpr l)
      ) ^ "]"
69    | Id(s) -> s
70    | Binop(e1, o, e2) ->
71        string_of_texpr e1 ^ " " ^
72        (match o with
73    Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/"
74        | Equal -> "==" | Neq -> "!="
75        | Mod -> "%"
76        | And -> "&&" | Or ->"||"
77        | Square -> "**"
78        | Less -> "<" | Leq -> "<="
79        | Greater -> ">" | Geq -> ">="
80        ) ^ " " ^ string_of_texpr e2
81    | Bool(x) -> if x = True then "true" else "false"
82    | Length(v) -> string_of_texpr v ^ ".length()\n"
83    | Access(v, e) -> string_of_texpr v ^ ".at(" ^ ( string_of_texpr e ) ^ ")\
      n"
84
85
86
87
88  let rec string_of_tstmt = function
89      texpr(texpr) -> string_of_texpr texpr ^ ""
90    | Var_Decl(tp, id) -> tp ^ " " ^ id ^ "\n"
91    | List_Decl(tp, id) -> "list " ^ tp ^ " " ^ id ^ "\n"
92    | Passign(v, e1, e) -> " " ^ string_of_texpr v ^ " = " ^ ( string_of_texpr
        e1 ) ^ "\n"
93    | Assign(v, e) -> "" ^ string_of_texpr v ^ " = " ^ ( string_of_texpr e )
94    | Append(v, e) -> string_of_texpr v ^ ".append(" ^ ( string_of_texpr e ) ^
      ")\n"
95    | Pop(v) -> string_of_texpr v ^ ".pop()\n"
96    | Remove(v, e) -> string_of_texpr v ^ ".remove(" ^ ( string_of_texpr e ) ^
      ")\n"
97    | Fcall(v, el) ->  v ^ "("^ (String.concat "," (List.map string_of_texpr
      el)) ^")\n"
98    | PrintXY(e1,e2) -> "printXY( " ^ string_of_texpr e1 ^ "," ^
      string_of_texpr e2 ^ ")\n"
99    | Print(e) -> "print " ^ string_of_texpr e ^ "\n"
100   | LineVar(e1,e2)-> "line (" ^ string_of_texpr e1 ^ "," ^ string_of_texpr
      e2 ^ ")" ^ "\n"
101   | LineRaw(e1,e2,e3,e4)-> "line ( (" ^ string_of_texpr e1 ^ "," ^
      string_of_texpr e2 ^ ")" ^ "," ^ "(" ^ string_of_texpr e3
102                               ^ "," ^ string_of_texpr e4 ^ ") )\n"
103   | LinePX(e1, e2, e3)-> "line ( ( " ^ string_of_texpr e1 ^ "," ^
      string_of_texpr e2 ^ ") ," ^ string_of_texpr e3 ^ ") \n"
104   | For(s1, e1, s2, body) -> "for " ^ string_of_tstmt s1 ^ " ; " ^
      string_of_texpr e1 ^ " ; " ^ string_of_tstmt s2 ^ ": \n"
105                               ^ ( String.concat "\n\t" (List.map
      string_of_tstmt body) )
```

```
106                                 ^ "\nend\n"
107     | While(e, body) -> "while " ^ string_of_texpr e ^ " :\n" ^ (String.concat
          "\n\t" (List.map string_of_tstmt body)) ^ "\nend\n"
108     | Ifelse(e, succ_tstmt, else_tstmt) -> "if " ^ string_of_texpr e ^ " :\n"
         ^ (String.concat "\n\t" (List.map string_of_tstmt succ_tstmt)) ^ "\nelse:\
         n" ^ (String.concat "\n\t" (List.map string_of_tstmt else_tstmt)) ^ "end\n
         "
109     | Return(texpr) -> "return " ^ string_of_texpr texpr ^ "\n"
110     | Notexpr -> ""
111     | Fdecl(f) -> string_of_fdecl f and
112     string_of_fdecl fdecl =
113         "fn " ^ fdecl.fname ^ "(" ^
114           ( String.concat ", " (List.map (fun s -> string_of_tstmt s) fdecl.
          args) ) ^
115           "):\n" ^
116         ( String.concat "" (List.map string_of_tstmt fdecl.body) ) ^
117         "\nend\n"
118
119   let string_of_program prog =
120     String.concat "\n" (List.map string_of_tstmt prog.funcs)
121     ^ "\n----------\n" ^
122     String.concat "\n" (List.map string_of_tstmt prog.main) *)
123
124
125
```

./src/test.py

```
import sys
import os

#If user gives a specific set of files from command line
if len(sys.argv)>1:
 testFiles = sys.argv[1:]
else:
 #Get all the files in the tests dir
 testFiles = os.listdir('./tests/')
 testFiles = [ x for x in testFiles if ( x[-3:]=='plt' and ( x[:4] in ['pass','fail'] ))]

nof = len(testFiles)

#passing and failing
passed = []
failed = []
i=0
print 'Starting the tests..'
#Proceed if make succeeds
for file in testFiles:
 #if i%(nof/20) == 0:
 sys.stdout.write('.')
 #For each test file perform the test. And print pass or failure
 runStr = './plt tests/' + file + ' 2> temp.out'
 #print 'Running for file : '+file +'\n'
 os.system(runStr)
 f = open('temp.out')
 s = f.readlines()
 f.close()
 if (len(s)>0 and file[:4]=='pass') or (len(s)==0 and file[:4]=='fail'):
   failed.append('**** FAILED for file '+file+'\n' + ' '.join(s) )
 else:
   passed.append( 'PASSED for --  '+file)
 i+=1
#Printing the results
print '\n---------- PASSED TESTS ----------'
for i in passed:
 print i
print '---------- FAILED TESTS ----------'
for i in failed:
 print i

print '---------- TESTS STATS----------'
print 'Passed : ' + str(len(passed))
print 'Failed : ' + str(len(failed))


```

./src/library/plots.plt

```
/*
    Method :
        rect(point xy, num h, num w)

    Return :
        void

    Description :
        Prints a rectangle, from point x,y with height h and width w The function rectangle supported as libray.

    Future Todo :
        Send hash, to set the properties of the graph like axis, grid, title etc
 */
fn rect(point a, num h, num w):
    line(a,(a[0]+w,a[1]))
    line(a,(a[0],a[1]+h))
    line((a[0]+w,a[1]),(a[0]+w,a[1]+h))
    line((a[0],a[1]+h),(a[0]+w,a[1]+h))
end

fn rectFill(point a, num h, num w, string color):
    num i
    point x
    point y
    string s

    /* Make a rectanle by drawing multiple lines */
    for i=0;i<w;i=i+1:
        x = (a[0]+i, a[1])
        y = (a[0]+i, a[1]+h)
        line( x , y , color )
    end
end

/*
    Method :
        drawAxes(num xorigin, num yorigin, num maxWidth, num maxHeight)

    Return :
        void

    Description :
        Draws cross axis from the passed origin point. Takes max width and max Height to plot lines

    Future Todo :
        Send hash, to set the properties of the graph like axis, grid, title etc
 */
fn drawAxes(num ox, num oy, num maxLength, num maxHeight):

    #Draw axes
    rect((ox,0), maxHeight, 3)
    rect((0,maxHeight-oy),3,maxLength)
end

/*
```

```
56      Method :
57          barGraph(<num list var>)
58
59      Return :
60          void
61
62      Description :
63          Prints a fitting barGraph for the given data. The function bargraph
        supported as libray. Users can write their own too. It takes the list of
        num and automatically fits it in the graph based on the data.
64      The user just needs to call the function with the data
65
66      Future Todo :
67          Send hash, to set the properties of the graph like axis, grid, title
        etc
68  */
69  fn barGraph(list num a):
70      num maxLength
71      num maxHeight
72      num maxDataHt
73      num maxDataLn
74      num gap
75      num scaleFactor
76      num padHz
77      num padVt
78      num barWidth
79      num i
80
81
82      #Setting the dimensions of the graph
83      maxLength = 640
84      maxHeight = 480
85
86      #Max ht
87      maxDataHt = a[0]
88      for i=0;i<a.length();i=i+1:
89          if a[i] > maxDataHt:
90              maxDataHt = a[i]
91          end
92      end
93
94      #max length
95      maxDataLn = a.length()
96
97      #padding
98      padHz = 10
99      padVt = 10
100
101     #bar graph settings 10% of the graph
102     gap         = 0.1 * (maxLength - padHz) / maxDataLn
103     barWidth    = 0.9 * (maxLength - padHz) / maxDataLn
104     scaleFactor = (maxHeight - padVt) / maxDataHt
105
106     #Draw the bars, scaled and with the gap
107     num x
108     x = padHz
109     for i=0;i<a.length();i=i+1:
110         #Drawing the bar
111         rect( (x,maxHeight- a[i]*scaleFactor) , a[i]*scaleFactor , barWidth
        )
```

```
112         #line( (x,maxHeight- a[i]*scaleFactor) , (x,maxHeight) )
113         #print x
114         x = x + barWidth + gap
115     end
116
117
118 end
119
120 /*
121     Method :
122         lineGraph(num xorigin , num yorigin , num maxWidth, num maxHeight)
123
124     Return :
125         void
126
127     Description :
128         Draws a line graph  based on the points give. To draw axes call it
    seperately
129
130     Future Todo :
131         Send hash, to set the properties of the graph like axis, grid, title
    etc
132 */
133 fn lineGraph(list num a):
134     num maxLength
135     num maxHeight
136     num maxDataHt
137     num maxDataLn
138     num gap
139     num scaleFactor
140     num padHz
141     num padVt
142     point p1
143     point p2
144     num i
145
146
147     #Setting the dimensions of the graph
148     maxLength = 640
149     maxHeight = 480
150
151     #Max ht
152     maxDataHt = a[0]
153     for i=0;i<a.length();i=i+1:
154         if a[i] > maxDataHt:
155             maxDataHt = a[i]
156         end
157     end
158
159     #max length
160     maxDataLn = a.length()
161
162     #padding
163     padHz = 10
164     padVt = 10
165
166     #Line graph settings 10% of the graph
167     gap         = (maxLength - padHz) / maxDataLn
168     scaleFactor = (maxHeight - padVt) / maxDataHt
169
```

```
170     #Draw the bars, scaled and with the gap
171     num x
172     num j
173     num y1
174     num y2
175     x = padHz
176 
177     if a.length() > 0:
178 
179         for i=1;i<a.length();i=i+1:
180             line( (x , maxHeight - a[ i - 1 ] * scaleFactor) , ( x + gap ,
    maxHeight- a[ i ]*scaleFactor) )
181 
182             #dots on the data points
183             rect ( (x - 2.5, maxHeight - a[ i - 1 ] * scaleFactor - 2.5) , 5
    , 5)
184             rect ( (x - 2.5, maxHeight - a[ i - 1 ] * scaleFactor - 2.5) , 5
    , 5)
185             x = x + gap
186 
187         end
188     end
189 end
190 
191 
```

./src/programs/bargraph.plt

```
fn rect(point a, num h, num w):
    num i
    num x
    num y
    point b
    /* Make a rectanle by drawing multiple lines */
    for i=0;i<w;i=i+1:
        x = a[0]
        y = a[1]
        line( (x+i , y), (x+i, y+h))
    end
end

/*
    Method :
        barGraph(<num list var>)

    Return :
        void
        Prints a fitting barGraph for the given data

    Description :
        The function bargraph supported as libray. Users can write their own too. It takes the list of num and automatically fits it in the graph based on the data.
    The user just needs to call the function with the data

    Future Todo :
        Send hash, to set the properties of the graph like axis, grid, title etc
 */
fn barGraph(list num a):
    num maxLength
    num maxHeight
    num maxDataHt
    num maxDataLn
    num gap
    num scaleFactor
    num padHz
    num padVt
    num barWidth
    num i


    #Setting the dimensions of the graph
    maxLength = 640
    maxHeight = 480

    #Max ht
    maxDataHt = a[0]
    for i=0;i<a.length();i=i+1:
        if a[i] > maxDataHt:
            maxDataHt = a[i]
        end
    end

    #max length
    maxDataLn = a.length()

```

```
57      #padding
58      padHz = 10
59      padVt = 10
60
61      #bar graph settings 10% of the graph
62      gap         = 0.1 * (maxLength - padHz) / maxDataLn
63      barWidth    = 0.9 * (maxLength - padHz) / maxDataLn
64      scaleFactor = (maxHeight - padVt) / maxDataHt
65
66      #Draw the bars, scaled and with the gap
67      num x
68      x = padHz
69      for i=0;i<a.length();i=i+1:
70          #Drawing the bar
71          rect( (x,maxHeight- a[i]*scaleFactor) , a[i]*scaleFactor , barWidth
    )
72          #line( (x,maxHeight- a[i]*scaleFactor) , (x,maxHeight) )
73          #print x
74          x = x + barWidth + gap
75      end
76
77
78  end
79
80  list num a
81  a = [11,4,25.6,10,12,50,10,30,5]
82  barGraph(a)
83
84
```

./src/programs/box.plt

```
1  fn rect(point a, num h, num w):
2      line(a,(a[0]+w,a[1]))
3      line(a,(a[0],a[1]+h))
4      line((a[0]+w,a[1]),(a[0]+w,a[1]+h))
5      line((a[0],a[1]+h),(a[0]+w,a[1]+h))
6  end
7
8  rect ( (100,100), 50, 20)
9
10
```

./src/programs/colorline.plt

```
1 point p1
2  point p2
3  p1 = (100,100)
4  p2 = (100,200)
5  line( p1 , p2 , "blue")
6
7
8
```

./src/programs/comicBookTemplate.plt

```
/*
    Method :
        rect(point xy, num h, num w)

    Return :
        void

    Description :
        Prints a rectangle, from point x,y with height h and width w The function rectangle supported as libray.

    Future Todo :
        Send hash, to set the properties of the graph like axis, grid, title etc
 */
fn rect(point a, num h, num w):
    line(a,(a[0]+w,a[1]))
    line(a,(a[0],a[1]+h))
    line((a[0]+w,a[1]),(a[0]+w,a[1]+h))
    line((a[0],a[1]+h),(a[0]+w,a[1]+h))
end

fn rectFill(point a, num h, num w, string color):
    num i
    point x
    point y
    string s

    /* Make a rectanle by drawing multiple lines */
    for i=0;i<w;i=i+1:
        x = (a[0]+i, a[1])
        y = (a[0]+i, a[1]+h)
        line( x , y , color )
    end
end

/*
    Method :
        drawAxes(num xorigin, num yorigin, num maxWidth, num maxHeight)

    Return :
        void

    Description :
        Draws cross axis from the passed origin point. Takes max width and max Height to plot lines

    Future Todo :
        Send hash, to set the properties of the graph like axis, grid, title etc
 */
fn drawAxes(num ox, num oy, num maxLength, num maxHeight):

    #Draw axes
    rect((ox,0), maxHeight, 3)
    rect((0,maxHeight-oy),3,maxLength)
end

/*
```

```
56      Method :
57          barGraph(<num list var>)
58
59      Return :
60          void
61
62      Description :
63          Prints a fitting barGraph for the given data. The function bargraph
        supported as libray. Users can write their own too. It takes the list of
        num and automatically fits it in the graph based on the data.
64      The user just needs to call the function with the data
65
66      Future Todo :
67          Send hash, to set the properties of the graph like axis, grid, title
        etc
68  */
69  fn barGraph(list num a):
70      num maxLength
71      num maxHeight
72      num maxDataHt
73      num maxDataLn
74      num gap
75      num scaleFactor
76      num padHz
77      num padVt
78      num barWidth
79      num i
80
81
82      #Setting the dimensions of the graph
83      maxLength = 640
84      maxHeight = 480
85
86      #Max ht
87      maxDataHt = a[0]
88      for i=0;i<a.length();i=i+1:
89          if a[i] > maxDataHt:
90              maxDataHt = a[i]
91          end
92      end
93
94      #max length
95      maxDataLn = a.length()
96
97      #padding
98      padHz = 10
99      padVt = 10
100
101     #bar graph settings 10% of the graph
102     gap         = 0.1 * (maxLength - padHz) / maxDataLn
103     barWidth    = 0.9 * (maxLength - padHz) / maxDataLn
104     scaleFactor = (maxHeight - padVt) / maxDataHt
105
106     #Draw the bars, scaled and with the gap
107     num x
108     x = padHz
109     for i=0;i<a.length();i=i+1:
110         #Drawing the bar
111         rect( (x,maxHeight- a[i]*scaleFactor) , a[i]*scaleFactor , barWidth
        )
```

```
112             #line( (x,maxHeight- a[i]*scaleFactor) , (x,maxHeight) )
113             #print x
114             x = x + barWidth + gap
115         end
116
117
118     end
119
120     /*
121         Method :
122             lineGraph(num xorigin , num yorigin , num maxWidth, num maxHeight)
123
124         Return :
125             void
126
127         Description :
128             Draws a line graph  based on the points give. To draw axes call it
        seperately
129
130         Future Todo :
131             Send hash, to set the properties of the graph like axis, grid, title
        etc
132     */
133     fn lineGraph(list num a):
134         num maxLength
135         num maxHeight
136         num maxDataHt
137         num maxDataLn
138         num gap
139         num scaleFactor
140         num padHz
141         num padVt
142         point p1
143         point p2
144         num i
145
146
147         #Setting the dimensions of the graph
148         maxLength = 640
149         maxHeight = 480
150
151         #Max ht
152         maxDataHt = a[0]
153         for i=0;i<a.length();i=i+1:
154             if a[i] > maxDataHt:
155                 maxDataHt = a[i]
156             end
157         end
158
159         #max length
160         maxDataLn = a.length()
161
162         #padding
163         padHz = 10
164         padVt = 10
165
166         #Line graph settings 10% of the graph
167         gap         = (maxLength - padHz) / maxDataLn
168         scaleFactor = (maxHeight - padVt) / maxDataHt
169
```

```
170     #Draw the bars, scaled and with the gap
171     num x
172     num j
173     num y1
174     num y2
175     x = padHz
176 
177     if a.length() > 0:
178 
179         for i=1;i<a.length();i=i+1:
180             line( (x , maxHeight - a[ i - 1 ] * scaleFactor) , ( x + gap ,
    maxHeight- a[ i ]*scaleFactor) )
181 
182             #dots on the data points
183             rect ( (x - 2.5, maxHeight - a[ i - 1 ] * scaleFactor - 2.5) , 5
    , 5)
184             rect ( (x - 2.5, maxHeight - a[ i - 1 ] * scaleFactor - 2.5) , 5
    , 5)
185             x = x + gap
186 
187         end
188     end
189 end
190 
191 fn comicBookTemplate(list num rowDetails, list num colDetails):
192     num maxHeight
193     num maxLength
194     num x
195     num y
196     num i
197     num j
198     num k
199     num h
200     maxHeight= 600
201     maxLength= 480
202     x=0
203     y=0
204     j = 0
205     for i=0;i<rowDetails.length();i=i+2:
206         x = 0
207 
208         #Border for the row
209         h = maxHeight * rowDetails[ i ]
210         rectFill((0,y), h, maxLength,"brown")
211         rectFill( ( 5 , y+5), h - 10, maxLength - 10, "white" )
212 
213         #Putting the columns
214         for k=0; k +1 < rowDetails[i + 1];k=k+1:
215             x = x + colDetails[j+k]*maxLength
216             rectFill(( x , y), h , 5 , "black" )
217             #printXY("*",( x - 20 , y+h/2))
218         end
219         j = j + k + 1
220         y = y+h - 5
221     end
222 
223 end
224 
225 list num rd
226 list num cd
```

```
#Give row height % and no of columns in that row
rd = [0.3, 2, 0.4, 1, 0.3, 2]

#Give column % wrt the whole width
cd = [0.3,0.7,1,0.7,0.3]

comicBookTemplate(rd, cd)
```

./src/programs/empireStates.plt

```
include plots
#Rectabgle with black border
fn rectWB(point p, num w, num h, string c):
    rectFill(p,w,h,c)
    rect(p,w,h)
end



rectWB((0,550),50, 300,"grey")

rect((65,135),465, 170)
rectWB((85,120),480, 130,"grey")
rectWB((90,120),480, 120,"white")

rectWB((100,100),500, 100,"grey")

#Centre row
rectWB((125,50),550, 50,"white")
rectWB((145,0),600, 10, "white")

#Pillars
rectWB((50,150),450, 70, "white")
rectWB((50,200),400, 70, "white")
rectWB((180,150),450, 70, "white")
rectWB((180,200),400, 70, "white")
```

./src/programs/forLoop.plt

```
num a
 for a = 0; a < 5; a = a + 1 :
 print a
 end


```

./src/programs/hello.plt

```
1 print "Hello brothers and sisters"
2
3
```

./src/programs/ifelse.plt

```
num a
 a = 5
 if a > 0 :
     print "A is positive"
 end
 else :
     print "A is negative"
 end

```

./src/programs/line.plt

```
1 line((50, 500), (500, 50))
2
3
```

./src/programs/lineGraph.plt

```
#include plots
 fn rect(point a, num h, num w):
     num i
     num x
     num y
     point b
     /* Make a rectanle by drawing multiple lines */
     for i=0;i<w;i=i+1:
         x = a[0]
         y = a[1]
         line( (x+i , y), (x+i, y+h))
     end
 end

 fn drawAxes(num ox, num oy, num maxLength, num maxHeight):

     #Draw axes
     rect((ox,0), maxHeight, 3)
     rect((0,maxHeight-oy),3,maxLength)
 end

 fn lineGraph(list num a):
     num maxLength
     num maxHeight
     num maxDataHt
     num maxDataLn
     num gap
     num scaleFactor
     num padHz
     num padVt
     point p1
     point p2
     num i


     #Setting the dimensions of the graph
     maxLength = 640
     maxHeight = 480

     #Max ht
     maxDataHt = a[0]
     for i=0;i<a.length();i=i+1:
         if a[i] > maxDataHt:
             maxDataHt = a[i]
         end
     end

     #max length
     maxDataLn = a.length()

     #padding
     padHz = 10
     padVt = 10

     #Line graph settings 10% of the graph
     gap         = (maxLength - padHz) / maxDataLn
     scaleFactor = (maxHeight - padVt) / maxDataHt

     #Draw the bars, scaled and with the gap
```

```
60      num x
61      num j
62      num y1
63      num y2
64      x = padHz
65
66      if a.length() > 0:
67
68          for i=1;i<a.length();i=i+1:
69              line( (x , maxHeight - a[ i - 1 ] * scaleFactor) , ( x + gap ,
    maxHeight- a[ i ]*scaleFactor) )
70
71              #dots on the data points
72              rect ( (x - 2.5, maxHeight - a[ i - 1 ] * scaleFactor - 2.5) , 5
    , 5)
73              rect ( (x - 2.5, maxHeight - a[ i - 1 ] * scaleFactor - 2.5) , 5
    , 5)
74              x = x + gap
75
76          end
77      end
78  end
79
80  list num a
81  a = [50,53, 55, 53, 50, 45, 25, 10, 5, 3, 0, 2, 4, 10, 25, 45,60, 75, 85,
    90,95, 98, 99, 100]
82  lineGraph(a)
83  drawAxes( 10,240,640,480)
84
85
86
87
88
```

./src/programs/list_access.plt

```
1 list num a
2  a = [1, 2, 3]
3  print a.at(0)
4  print a[0]
5
```

./src/programs/list_decl.plt

```
list num a
 list string b
 list bool d
 a = []
 a = [1]
 a = [1,2,3]
 b = []
 b = ["c"]
 b = ["a","b","c"]
 d = []
 d = [true]

 d = [true, false, true]
 a.append(6)


```

./src/programs/list_operations.plt

```
1  list num a
2   list bool b
3   a=[1,2,3,4,5,6]
4   b=[]
5   a.append(8)
6   a.pop()
7   a.remove(4)
8   a.at(3)
9   a[4]
10  a.length()
11  num x
12  x = 1 + 4
13
14
```

./src/programs/plot1.plt

```
num i
 i = 0
 while i <= 500:
  line((50,i+50), (550, 500-i+50))
  line((50+i,50),(550-i,550))
  i = i + 10
 end


```

./src/programs/print.plt

```
num a
 a = 5 + 10
 print a
 print "a"


```

./src/programs/printXY_sample.plt

```
num i
 for  i=0;i<10;i=i+1:
      printXY("Ranjith",(100,100+20*i))
 end


```

./src/programs/rectangle.plt

```
#In future it'll be from library
 fn rect(point a, num h, num w):
     num i
     num x
     num y
     point b
     /* Make a rectanle by drawing multiple lines */
     for i=0;i<w;i=i+1:
         x = a[0]
         y = a[1]
         line( (x+i , y), (x+i, y+h))
     end
 end

 point a
 a = (100,100)

 #Rectangles with increasing width, nice pattern too
 rect(a, 200, 10)
 rect((100+15,100), 200, 50)
 rect((100+70,100), 200, 100)
 rect((100+175,100), 200, 200)
 rect((100+380,100), 200, 500)
 rect((100+885,100), 200, 500)


```

./src/programs/rectFill.plt

```
fn rect(point a, num h, num w, string color):
    line(a,(a[0]+w,a[1]))
    line(a,(a[0],a[1]+h))
    line((a[0]+w,a[1]),(a[0]+w,a[1]+h))
    line((a[0],a[1]+h),(a[0]+w,a[1]+h))
end
fn rectFill(point a, num h, num w, string color):
    num i
    point x
    point y
    string s

    /* Make a rectanle by drawing multiple lines */
    for i=0;i<w;i=i+1:
        x = (a[0]+i, a[1])
        y = (a[0]+i, a[1]+h)
        line( x , y , color )
    end
end

rectFill ( (100,100), 50, 20,"blue")
```

./src/programs/simple_plot.plt

```
1 line((50, 500), (50, 50))
2  line((50, 500), (600, 500))
3  line((50, 500), (500, 50))
4
5
```

./src/programs/sine.plt

```
#In future it'll be from library
 fn rect(point a, num h, num w):
     num i
     num x
     num y
     point b
     /* Make a rectanle by drawing multiple lines */
     for i=0;i<w;i=i+1:
         x = a[0]
         y = a[1]
         line( (x+i , y), (x+i, y+h))
     end
 end

 num i
 num ht
 num flag

 #Initial
 flag = 1

 # Trying to print a sine wave with rectange function
 for i=0;i<1000;i=i+1:
     rect((100+5*i,100+ht),200,4)

     ht = ht + flag*2
     #Sine switch
     if ht==100 or ht==0:
         flag = flag* -1
     end
 end


```

./src/programs/sum.plt

```
num a
 a = -5 + 10
 print a


```

./src/programs/vdecl.plt

```
1 string a
2  num b
3  bool c
4
5
```

./src/programs/while.plt

```
1 num a
2  a = 0
3  while a < 10:
4  print a
5  a = a + 1
6  end
7
8
```

./src/tests/fail$_{b}ool_{a}ssign_{n}um_{f}loat.plt$

```
1 bool a
2  a = 5.5
3
4
```

./src/tests/fail$_{b}ool_{a}ssign_{n}um_{i}nt.plt

```
1 bool a
2  a = 5
3
4
```

./src/tests/fail$_{b}ool_{a}ssign_{s}tring.plt

```
1 bool a
2  a = "hi"
3
4
```

./src/tests/fail$_{b}ool_{s}um_{n}um.plt

```
1 bool a
2  a = true + 5
3
4
```

./src/tests/fail$_{b}ool_{s}um_{p}oint.plt

```
1 bool a
2  point b
3  a = true + b
4
5
```

./src/tests/fail$_{b}ool_{s}um_{s}tring.plt

```
1 bool a
2  a = true + "hi"
3
4
```

./src/tests/fail$_{f}or_{c}olon.plt

```
num a
 for a = 0; a < 5; a = a + 1
 print a
 end


```

./src/tests/fail_for_empty.plt

```
num a
 for :
 end


```

./src/tests/fail$_{f}or_{e}nd.plt

```
num a
 for a = 0; a < 5; a = a + 1 :
 print a


```

./src/tests/fail$_{for}{}_{missing}{}_{1}$.plt

```
num a
 for a < 5; a = a + 1 :
 print a
 end


```

./src/tests/fail$_{for}{}_{missing}{}_{23}$.plt

```
num a
 for a = 0; :
 print a
 end


```

./src/tests/fail$_{f}or_{m}issing_{3}.plt

```
num a
 for a = 0; a < 5; :
 print a
 end


```

./src/tests/fail$_{f}or_{m}issing_{t}wo.plt

```
num a
 for a = 0;   a = a + 1 :
 print a
 end


```

./src/tests/fail_include_lib.plt

```
list num a
 a = [100,80,95,66,70]
 barGraph(a)


```

./src/tests/fail$_{i}nvalid_{f}unction_{c}all.plt$

```
fn plt():
    print "plt"
end
fn pltr():
    print "plt"
end
fn plt():
    print "plt"
end
pltt()


```

./src/tests/fail$_{l}ine_{b}ool.plt

```
1 bool a
2  a = true
3  line((a,a), (a,a))
4
5
```

./src/tests/fail$_{l}ine_{b}ool_{n}um.plt

```
1 bool a
2  a = true
3  line((5,a), (10, a))
4
```

./src/tests/fail$_{line_{num_{pt.plt}}}$

```
1 point x
2  x =  (500 , 500 )
3  line(10, x)
4
5
```

./src/tests/fail$_{l}ine_{n}um_{s}tring.plt$

```
1 string a
2  a = "5"
3  line((5,a), (10, a))
4
```

./src/tests/fail$_{l}ine_{p}t_{n}um.plt$

```
1 point x
2  x =  (500 , 500 )
3  line(x, 10)
4
5
```

./src/tests/fail$_{l}ine_{s}tring.plt$

```
1 string a
2  a = "5"
3  line((a,a),(a,a))
4
5
```

./src/tests/fail$_{num}$_{assign}$_{bool}$.plt

```
1 num a
2  a = true
3
4
```

./src/tests/fail$_{num_assign_string.plt}$

```
num a
  a = "hi"


```

./src/tests/fail$_{n}um_{s}um_{b}ool.plt$

```
num a
 a = 5 + true


```

./src/tests/fail$_{n}um_{s}um_{p}oint.plt$

```
point a
 num b
 b = 5 + a


```

./src/tests/fail$_{n}um_{s}um_{s}tring.plt$

```
num a
  a = 5 + "hi"


```

./src/tests/fail$_{p}oint_{d}ecl_{b}ool.plt$

```
1 point a
2  a = (true, 5.5)
3
4
```

./src/tests/fail$_{p}oint_{d}ecl_{b}ool_{v}ar.plt$

```
1 bool a
2  a = true
3  point b = (a,a)
4
5
```

./src/tests/fail$_{p}oint_{d}ecl_{s}tring.plt$

```
point a
 a = ("5.5", 5.5)


```

./src/tests/fail$_{p}oint_{d}ecl_{s}tring_{v}ar.plt$

```
1 string a
2  a = "5"
3  point b
4  b = (a,a)
5
6
```

./src/tests/fail$_s tring_a ssign_b ool.plt

```
1 string a
2  a = true
3
4
```

./src/tests/fail$_{s}tring_{n}um_{f}loat_{a}ssign.plt$

```
1 string a
2  a = 5.5
3
4
```

./src/tests/fail$_string_num_int_assign.plt$

```
1 string a
2  a = 5
3
4
```

./src/tests/fail$_s tring_s um_b ool$

```
1 string a
2  a = "hi" + bool
3
4
```

./src/tests/fail$_string_sum$_num$_um.plt

```
1 string a
2  a = "hi" + 5
3
4
```

./src/tests/fail$_s tring_s um_p oint.plt

```
1 point a
2  string b
3  b = "5" + a
4
5
```

./src/tests/fail$_w$*hile*$_c$*olon.plt*

```
num a
 a = 1
 while (a < 10)
 print a
 a = a + 1
 end


```

./src/tests/fail$_{w}hile_{c}ond_{s}tring.plt$

```
1 string a
2  a = "hi"
3  while (a):
4  print a
5  end
6
7
```

./src/tests/fail$_w hile_e nd.plt

```
num a
 a = 1
 while (a < 10):
     print a
     a = a + 1


```

./src/tests/pass_bool_assign.plt

```
1 bool a
2  a = true
3
4
```

./src/tests/pass_bool_decl.plt

```
1 bool a
2  bool aaa
3  bool ab12pong
4
5
```

./src/tests/pass_bool_unassigned.plt

```
1 bool a
2  print a
3
4
```

./src/tests/pass_ol.plt

```
num a

```

./src/tests/pass_for_regular.plt

```
num a
 for a = 0; a < 5; a = a + 1 :
 print a
 end


```

./src/tests/pass_function_list_param.plt

```
fn printList(list num a):
      num j
      for j=0; j<5; j=j+1 :
      print a[j]
      end
end

list num a
a = [1,2,3,4,5,6]
printList(a)


```

./src/tests/pass$_{function_n p_n r.plt}$

```
#no parameter no return
 fn addOne():
      num a
      a = 3
 end

 addOne()
 num a
 a = 5
 a = a*5
```

./src/tests/pass$_{function_{pn}}$r.plt

```
#1+ parameter no return
 fn addOne(num a, num b, num c):
      a = a + 1
      b = b + 1
      c = c + 1
      print (a+b+c)
 end

 addOne(0,0,0)
 num a
 a = 5
 a = a*5
```

./src/tests/pass_function_two_fs.plt

```
fn addOne():
      num a
      a = 3
      a = 3
end
fn addTwo(num aa, num b):
      num a
      a = 5
      print aa + b
end
fn addThree(num aa):
      num a
      a = 3
      print aa + a
end
fn addFour(num aa, num b):
      num a
      a = 4
      b = a + aa
      print b
end

addOne()
addTwo(3,3)
addThree(3)
addFour(3,0)

num a
a = 5
a = a*5
```

./src/tests/pass$_i$*f.plt*

```
1 bool a
2  a = true
3  if a:
4      print "true"
5  end
6
7
```

./src/tests/pass$_i$*felse.plt*

```
num a
 a = 7
 if a:
     print "true"
     if a:
         print "its still true"
     else:
         print "there is something wrong"
     end
 end


```

./src/tests/pass_include_lib.plt

```
include plots
 list num a
 a = [100,80,95,66,70]
 barGraph(a)


```

./src/tests/pass_line_int_float.plt

```
1 line((5.5,5),(100,100.5))
2
3
```

./src/tests/pass_line_nums.plt

```
num a
 a = 5
 num b
 b = 100.5
 line((a,a),(b,b))


```

./src/tests/pass$_{l}ine_{n}ums_{p}t.plt

```
point x
 x =  (500 , 500 )
 line((10, 10), x)


```

./src/tests/pass$_{l}ine_{p}oint_{n}ums.plt$

```
1 point b
2  b = (100, 100)
3  line((5,5), b)
4
5
6
```

./src/tests/pass_line_points

```
1 point a
2  a = (5,5)
3  point b
4  b = (100,100)
5  line(a, b)
6
7
```

./src/tests/pass_line_pt_nums.plt

```
point x
 x =  (500 , 500 )
 line(x,( 10, 10))


```

./src/tests/pass_linegraph.plt

```
include plots

 list num a
 a = [50,53, 55, 53, 50, 45, 25, 10, 5, 3, 0, 2, 4, 10, 25, 45,60]
 lineGraph(a)


```

./src/tests/pass_list_append.plt

```
1 list num a
2  a = [3, 5, 7, 9]
3  a.append(20)
4
5
```

./src/tests/pass_list_iterator.plt

```
1 list num aa
2  aa = [1,2,3,4,5,6]
3  num j
4  for j=0; j<5; j=j+1 :
5  print aa[j]
6  end
7
8
```

./src/tests/pass_list_length.plt

```
num a
 list num b
 b = [1,2,3,4]
 a = b.length()
 print a


```

./src/tests/pass_list_pop.plt

```
1 list num a
2  a = [10, 20, 30, 40]
3  a.pop()
4
5
```

./src/tests/pass_list_remove.plt

```
1 list num a
2  a = [10, 20, 30, 40]
3  a.remove(1)
4
5
```

./src/tests/pass_multiline_comment.plt

```
/* yo
   this
   is
   * MULTI LINR COMMENT
   num a
   print b
   num num num string;
   these are not parsed
 */

 print "Hello World"
 /* Simple Multi
    Line comment
    hi */ print "hello again"

 /* backk */


```

./src/tests/pass_num_assign_float.plt

```
num a
 a = 5.5


```

./src/tests/pass$_{n}um_{a}ssign_{i}nt.plt$

```
num a
  a = 5


```

./src/tests/pass$_num_decl.plt$

```
num a
 num abc
 num A123
 num a0


```

./src/tests/pass_num_diff.plt

```
num a
  a = 10 - 5


```

./src/tests/pass$_num$_sum.plt

```
num a
  a = 10 + 5


```

./src/tests/pass$_{n}um_{u}nassigned.plt$

```
num a
 print a


```

./src/tests/pass$_{p}oint_{d}ecl_{f}loat.plt$

```
1 point a
2  a = (5.5, 5.5)
3
4
```

./src/tests/pass$_{p}oint_{d}ecl_{i}nt.plt$

```
1 point a
2  a = (5, 5)
3
4
```

./src/tests/pass$_{p}oint_{d}ecl_{n}um.plt$

```
num a
 a = 5
 point b
 b = (a,a)
 num aA01a

 a = 5
 point b007Bond
 b007Bond = (aA01a, aA01a)


```

./src/tests/pass$_{p}oint_{d}ecl_{n}um_{s}um.plt$

```
num a
 a = 5
 point p
 p = (a+5, a+15)


```

./src/tests/pass$_{p}rint_{b}ool.plt$

```
1 bool a
2  a = true
3  print a
4
5
```

./src/tests/pass$_{p}rint_{n}um.plt$

```
num a
 a = 5
 print a


```

./src/tests/pass_print_point.plt

```
print (1,2)


```

./src/tests/pass$_{p}rint_{s}tring.plt$

```
1 string a
2  a = "hi"
3  print a
4
5
```

./src/tests/pass_printXY.plt

```
num i
 for i=0;i<10;i=i+1:
     printXY("Ranjith",(100,100+20*i))
 end


```

./src/tests/pass_*return_stmt.plt*

```
num a
 a = 8
 if a > 5:
     return 0
 else:
     print "something wrong"
 end


```

./src/tests/pass$_{s}ingle_{l}ine_{c}omment.plt$

```
num a

 # this is a single line comment


```

./src/tests/pass$_{s}tring_{a}ssign.plt$

```
1 string a
2  a = "hi"
3
4
```

./src/tests/pass$_s$*tring*$_d$*ecl.plt*

```
1 string a
2  string abc
3  string A123
4  string a0
5
6
```

./src/tests/pass$_{s}tring_{u}nassigned.plt$

```
1 string a
2  print a
3
4
```

./src/tests/pass$_w$*hile.plt*

```
num a
 a = 1
 while (a < 10):
 print a
 a = a + 1
 end
```

./src/tests/pass$_{w}hile_{c}ond_{b}ool.plt$

```
num a
 a = 1
 while (((a<5))):
      print a
      a = a + 1
 end
```

./src/tests/pass$_{w}hile_{c}ond_{n}um.plt$

```
num a
 a = 1
 while (a):
      print a
      a = a + 1
      if a==5:
           a=0
      end
 end


```

./src/tests/test$_h$*ello.plt*

```
1 print "Hello world"
2
3
```

./src/tests/test$_{s}um.plt$

```
num a
 a = -5 + 10
 print a


```

./src/tests/test$_v$*decl.plt*

```
1 string a
2  num b
3  bool c
4
5
```