**Geo**

A geometric solution language

Qi Wang (qw2197), Yuechen Zhao (yz2877)

Zichen Chao (zc2321), Ziyi Luo (zl2471)

Final Report

December 17, 2014

# Contents

Chapter 1 – Introduction 4

1.1 Motivation 4

1.2 Key Features 4

1.2.1 Geometric Specific 4

1.2.2 Dynamic Movement 4

1.2.3 Complete Programming Functionality 4

Chapter 2 - Language Tutorial 5

2.1 Basic Structure of a Geo Program 5

2.2 Built-in Data Types and Variable Assignment 5

2.3 Control Structures 5

2.4 Writing and Calling a Function 6

2.5 Advanced Date Types 6

2.6 Geometric data types 6

2.7 Defining graphs 7

Chapter 3 - Language Reference Manual 8

3.1 Lexical Elements 8

3.1.1 Seperators 8

3.1.2 Comments 8

3.1.3 Identifiers 8

3.1.4 Keywords 9

3.1.5 Constants 9

3.1.6 Separators 10

3.1.7 Operators 10

3.2 Data Types 11

3.2.1 Basic Types 11

3.2.2 Geometric Types 12

3.2.3 list 13

3.2.4 model 14

3.3 Expressions 14

3.3.1 Variables 14

3.3.2 Presets 15

3.4 Functions 15

3.4.1 Function Declarations 16

3.4.2 Function Definitions 16

3.4.3 Calling Functions 16

3.5 Compound Statements 17

3.5.1 The if Statement 17

3.5.2 The while Statement 17

3.5.3 The for Statement 17

3.5.4 The run Statement 17

3.6 STD Library Reference 18

Chapter 4 - Project Plan 21

4.1 Project Management 21

4.1.1 Planning 21

4.1.2 Specification 21

4.1.3 Development 21

4.1.4 Testing 21

4.2 Programming Style Guide 22

4.3 Project Timeline 22

4.4 Roles and Responsibilities 23

4.5 Software Development Environment 23

4.6 Project Log 23

Chapter 5 - Architectural Design 24

5.1 Block Diagram 24

5.2 Components 24

5.2.1 Scanner 24

5.2.2 Parser and AST 24

5.2.3 Semantic Check and SAST   24

5.2.4 Code Generator   25

Chapter 6 - Test Plan 26

Chapter 7 - Lessons Learned 27

7.1 Qi Wang 27

7.2 Zichen Chao 27

7.3 Yuechen Zhao 27

7.4 Ziyi Luo 27

Chapter 8 – Appendix 29

# Chapter 1 – Introduction

Geo is an objective-orient geometric solution language that enables students, physicists, mathematicians, as well as other geometry-interested professionals to solve spatial problems efficiently. It involves functionalities that computes relationships among geometric figures, including lines, circles, and rectangles, etc. Users are able to define figures, set their moving patterns, and perform various analysis (including static and dynamic analyses) of the interacting figures. If desired, a user can also change the moving pattern or shape of objects.

## 1.1 Motivation

Many complex problems can be solved with graph beautifully, however, most standard programming languages do not have convenient tools for creating and manipulating graphs, especially dynamic graphs. So we decide to design a simple while powerful graph oriented programming language, Geo. It provides built-in support for graph, so users can define graphs in an intuitive way. In addition, the most important part about Geo is that graphs can move and interact, allowing dynamic analysis with a user-friendly interface.

## 1.2 Key Features

### 1.2.1 Geometric Specific

We provide an intuitive way for declaring graphs, defining relationships and make calculations. Users can declare a line by specifying the start point and end point. Users can also calculate the intersecting point of a line and a circle by a single command.

### 1.2.2 Dynamic Movement

Users can define moving objects in Geo, and see the animations directly through our panel interface. Dynamic analysis can be done easily with run statement, which will be explained in detail in the Language Tutorial part.

### 1.2.3 Complete Programming Functionality

Geo also has complete functionality as a normal programming language, simple algorithms such as Fibonacci number can be expressed in Geo in a recursive or iterative fashion; more complicated algorithms such as Dijkstra's shortest path algorithm can also be expressed.

# Chapter 2 - Language Tutorial

## 2.1 Basic Structure of a Geo Program

The hello world example of Geo is provided below, which contains a simplest function which takes in no arguments and only executes print.

// hello word

function hello(): void:

print("Hello World!");

end

## 2.2 Built-in Data Types and Variable Assignment

Geo supports the following primitive types: int, float, bool, char and string. The code below illustrates on how to assign values for built-in data types. One thing to notice is that we do not need to specify the types of variables because Geo knows types implicitly.

x = 10;

c = ’a’;

s = "hello";

flag = true;

## 2.3 Control Structures

The code below shoes the usage of if-else statement, for loop and while loop. It should print -1, h, e, l, l, o, 0 successively.

a = 0;

if(a>0):

a = a + 1;

else:

a = a - 1;

end

print(a);

for i in "hello":

print(i);

end

while(a<0)

a = a + 1;

print(a);

end

## 2.4 Writing and Calling a Function

The function declaration consists of the function keyword, function name, input arguments and their types, and finally a return type followed by a colon. A Geo program has no entry function, so main() is not necessary. The function body is all the statements above the end keyword. Below is an example of a recursive gcd algorithm that illustrates the usage of function.

function gcd(a:int, b:int): int:

while(a != b):

if(a>b):

a = a - b;

else:

b = b - a;

end

end

return a;

end

## 2.5 Advanced Date Types

For all primitive types, Geo has a corresponding array type. To declare an array, the size of the array is explicitly written in “[ ]” on the left side. The following example illustrates that, a does not have an associated data type until it is initialized, and its type cannot be changed then.

a[3];

a = [0, 0, 0];

## 2.6 Geometric data types

Geo supports geometric types including dot, line, polygons and circle. For geometric type, the declaration is quite different from the basic type variable declaration. For example, the declaration and initialization of a dot is shown as follow:

dot1 = [1.5,4.0];

dot2 = [3.0,4.0];

line1 = [dot1, dot2];

A dot has a x coordinate value and a y coordinate value which should be defined simultaneously, and a line has a start point and an end point. More on this are in chapter 3.

## 2.7 Defining graphs

A typical Geo program of graphs includes:

panel presets;

function declaration and definition;

geometric shape declaration and initialization;

runset declaration and initialization;

run statement description.

An example follows the above structure description is shown:

//panel presets

@panel panel\_demo

@mode figure

//function declaration and definition

function print\_dot\_list(dots[]:dot):void:

for d in dots:

print(d.getX(),' ',d.getY(),'\n');

end

end

//geometric shape declaration and initialization

line1 = line(2.0,3.0);

circle1 = circle([3,4], 5);

//runset declaration and initialization

line1.setRunstep(-0.5,'a');

circle1.setRunstep(0.1,'b');

rs = runset(50, line1, 'a', circle1, 'b');

//run statement description

run rs:

set = line1.intersect(circle1);

if (!set.empty())

print\_dot\_list(set);

end

@end

This example defines a function called print\_dot\_list to print every dot’s coordinates in a dot set using for loop. Then it declares a line and a circle, and they are set to move dynamically through setRunstep which lets objects move one step per second.

# Chapter 3 - Language Reference Manual

## 3.1 Lexical Elements

This section specifies the lexical elements of Geo programming language.

### 3.1.1 Seperators

Geo compiler regards the following char as the characters to separate tokens:

' '

'\t'

'\n'

At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs. Arbitrary combination of such characters to separate tokens is permitted for the Geo compiler will automatically ignored such characters when analyzing the program.

### 3.1.2 Comments

There are two kinds of comments in Geo:

Multiple-line comment: all the text in /\* comments \*/ is ignored (as in C and Java):

/\* text \*/

Single-line comment: all the text after // to the end of the line is ignored (as in Java):

// text

### 3.1.3 Identifiers

An identifier consists of a sequence of letters, digits and '\_', where the first character must be a letter. Identifiers are case-sensitive in Geo.

The following identifiers are legal:

abc

a7G

Foo

a\_b

The following identifiers are illegal:

7a

\_a

### 3.1.4 Keywords

The following identifiers are reserved as the keywords and cannot be used as identifiers.

bool

break

char

const

dot

elif

else

end

float

for

function

if

import

in

int

list

model

return

run

string

submodel

void

while

### 3.1.5 Constants

#### 3.1.5.1 Integer Constants

An integer constant is a sequence of digits starts with an optional positive/negative sign.

0

1

20

+210

-15

#### 3.1.5.2 Float constants

A float constant consists of an integer part, a decimal part, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

1.0

.5

2e3

5e-5

2.5e+10

#### 3.1.5.3 String Constants:

A string constant consists of a sequence of characters enclosed in double quotes.

"Hello world!"

"1234567890"

"Name\tID\tScore\n"

#### 3.1.5.4 Mathematical Constants

Mathematical constants include some frequently used parameters such as:

PI = 3.14159

### 3.1.6 Separators

Several characters are used as the separators:

( ) { } [ ] ; , .

Note that ';' denotes the end of a sentence. '.' is the access operator.

### 3.1.7 Operators

Following characters are regarded as the operators by Geo compiler:

|  |  |  |
| --- | --- | --- |
| Operator | Usage | Associativity |
| = | Assignment | Right |
| == | Equal to | - |
| != | Unequal to | - |
| > | Greater | - |
| >= | Greater or equal to | - |
| < | Less | - |
| <= | Less or equal to | - |
| & | Logic AND | - |
| | | Logic OR | - |
| ! | Logic NOT | Right |
| + | Addition | Left |
| - | Subtraction | Left |
| \* | Multiplication | Left |
| / | Division | Left |
| . | Access | Left |
| ^ | Exponentiation | Left |
| % | Modulo | Left |

The precedence of operators is as follows:

**.**

**\* / % ^**

**+ -**

**> >= < <=**

**!**

**& |**

**== !=**

**=**

## 3.2 Data Types

This chapter introduces all standard data types in Geo.

### 3.2.1 Basic Types

Basic Types includes int, float, bool, char and string.

For integer type int, the data ranges from -2147483648 to 2147483648.

For floating point type float, the data ranges from about -3.4E+38 to +3.4E+38.

For boolean type bool, it has two optional value true and false.

For character type char, the data ranges from NUL to DEL, that is, from 0 to 127.

For character string type string, the data consists of zero or more characters enclosed in double quotes. string is a powerful data type with several build-in functions in Geo STD library. Please refer to Geo STD library in Chapter 7 for more information about string.

### 3.2.2 Geometric Types

Geometric types includes geometric control type runset and geometric shapes include dot, line, polygons and circle.

#### 3.2.2.1 runset

Geometric control type runset is a data type to be recognized by the keyword run. In Geo STD library, runset is defined as:

model runset: runset(times\_of\_run:int,

g1:geometric\_shape, run\_para\_g1:char, ...);

times\_of\_run denotes the times that the run statement will execute. Note that the input of runset is not limited. After the first parameter times\_of\_run, the definition of runset allows unlimited **pairs** of geometric\_shape and run\_para\_g1. The parameter of the geometric instant (i.e., run\_para\_g1) must be set with a runstep using setRunstep function before adding to the runset. Otherwise, compilers will raise parameter\_unset error. For more build-in functions of runset, please refer to Geo STD library in Chapter 7.

#### 3.2.2.2 dot

Geometric shape dot represents a dot in panel. In Geo STD library, dot is defined as:

model dot: dot(x:float, y:float);

dot is a very special geometric type. The instantiation of a dot is quite different from other geometric shapes. Please refer to Section 4.1 for more information. For more build-in functions of dot, please refer to Geo STD library in Chapter 7.

#### 3.2.2.3 line

Geometric shape line represents a line (either finite or infinite long) in panel. In Geo STD library, line is defined as:

model line:

line(a:float,b:float);

line(dot1:dot,dot2:dot);

line(a:float,b:float,endpointx1:float,endpointx2:float);

line(dot1:dot,dot2:dot,endpointx1:float,endpointx2:float);

Any of the above four definitions can be used to initialize a line object. endpointx1 and endpointx2 are the x-coordinates of two endpoints of a line. For more build-in functions of line, please refer to Geo STD library in Chapter 7.

#### 3.2.2.4 polygons

Geometric shape polygons is a set of different shapes. Polygons has two special cases: Triangle Tri and Rectangle Rec. Polygons is defined as:

model polygons: polygons(num\_of\_apex:int,apex[]:dot);

Note that apex[] is a list of dots. For list, please refer to Section 3.3. Note that the length of apex must be exactly as num\_of\_apex and the apexes in the list should be in a reasonable order (i.e., either clockwise or counterclockwise). For more build-in functions of polygons, please refer to Geo STD library in Chapter 7.

#### 3.2.2.5 circle

Geometric shape circle represents a circle in panel. In Geo STD library, circle is defined as:

model circle: circle(center:dot,radius:float);

A circle is defined with a center coordinate and a radius. For more build-in functions of circle, please refer to Geo STD library in Chapter 7.

### 3.2.3 list

list is a set of a certain type of objects. A list contains a number of variables. The number of variables may be zero, in which case the list is said to be empty.

A list named listdemo with length of length\_of\_list and type of t is defined as:

listdemo[length\_of\_list]={t1:t,t2:t, … , tlength\_of\_list:t};

Syntax to get the nth element in listdemo is simple: listdemo[n-1];

For more build-in functions of list. For all lists, Geo offers several build-in functions shown as bellow:

function empty():bool;

function length():int;

function insert(ele:type\_in\_list);

function remove(order:int);

### 3.2.4 model

model is a keyword that allows users to define their own data type. Typically a model is built as follow:

model modelname:

variable declaration;

model initialization function;

function declaration;

end

model initialization function is a special function without return value and function keyword. The name of the initialization function is the same as name of the model.

## 3.3 Expressions

### 3.3.1 Variables

Variable consists of two parts: name and value. Name of a variable must be a legal identifier mentioned in Section 2.2. value of a variable depends on the data type of the variable.

#### 3.3.1.1 Basic type variable declaration and initialization

For basic types, the variable declaration and initialization is quite straightforward:

//Initialize an int named i with value equals 2

i = 2;

//Initialize a float named afloat with value equals 3e-5

afloat = 3e-5;

//Initialize a bool named judge with value equals true

judge = true;

//Initialize a char named c with value equals 'c'

c = 'c';

//Initialize a string named str with value equals "str"

str = "str";

Variables declaration without initialization is also allowed in Geo, the declaration rule is identifier:data\_type:

i:int;

afloat:float;

judge:bool;

c:char;

str:string;

#### 3.3.1.2 Geometric type variable declaration and initialization

For geometric type, the declaration is quite different from the basic type variable declaration. To declare a dot, the declaration and initialization are shown as follow:

dot1 = [1.5,2.2];

A dot is declared with two float parameters within a pair of square brackets. For other geometric types, the declaration and initialization examples are shown as follow:

//Declare a line y=3x+2 with x in [0,5.5].

line1 = line(3.0,2.0,0,5.5);

//Declare a circle (x-5)^2+(y-10)^2 = 6.2^2

circle1 = circle([5,10],6.2);

//Declare a pentagon with apex [0,0][2,0][2,2][1,5][0,3]

apex = {[0,0],[2,0],[2,2],[1,5],[0,3]};

pentagon = polygons(5,apex);

//Declare a line without initialization.

line2:line;

//Declare a polygon without initialization.

pentagon:polygon;

### 3.3.2 Presets

Presets occur at the beginning of the Geo program (except @end). Basic analyzing environment is set in presets. Each preset begins with an @ sign.

@panel panelname (essential) defines a panel with a legal identifier panelname.

@mode workingmode (optional) defines the mode that the program will be executed in. workingmode could be either console or figure. By default it works in console mode.

@co cosystem (optional) defines the coordinate system to be either cartesian or polar. By default it works in cartesian coordinate system.

@end (essential) indicates the boundary of a specific panel.

## 3.4 Functions

This chapter introduces functions in Geo. Functions are the key of extending the usage of models and solving geometric problems. Geo provides a powerful STD library for geometric problem solution while most of the methods in STD library are encapsulation in functions.

### 3.4.1 Function Declarations

A function is declared as the following rule:

function func\_name(parameter:parameter\_type,...):return\_type;|

Here function is the keyword for function declaration. func\_name is the identifier for this function. input parameters must be in the form of identifier:data\_type. return\_type is the type of the return value. If the function does not have a return value, :return\_type part should be :void.

Note that when list is used as the return value of a function, the return\_type is shown as : datatype[]. For example, when return value is a float list, the return\_type is float[].

### 3.4.2 Function Definitions

A typical function definition is shown as below. Note that keyword const in the first line of the function definition is an *optional* word; :return\_val\_type and return value(s); is not needed when the function do not have a return value(s);

function func\_name(para:para\_type const,...):return\_val\_type:

local\_variable declaration and initialization;

function operations;

return value(s);

end

Note that, local variables are available only in the domain of a function. All local variables will be terminated as the function ends. All parameters of functions are reference parameters in default, which means that if the input parameters are modified in the function, the parameter will keep that change after the execution of the function. The keyword const is required to avoid modifying input parameters and is strongly recommended to add in if needed.

### 3.4.3 Calling Functions

A function is called as the following rule:

val = func\_name(para1,para2, ...);

As for the functions in models. The function is called as:

val = model\_instant\_identifier.func\_name(para1,para2, ...); |

## 3.5 Compound Statements

### 3.5.1 The if Statement

A if-elif-else control flow follows the following rule:

if (condition1):

statement1;

elif (condition2):

statement2;

else:

statement3;

end

if, elif, else and end are keywords for the control flow. Condition 1 through condition 2 defines conditions to enter the if/elif cases, and statements 1 through 3 defines the statements to execute in three corresponding cases.

### 3.5.2 The while Statement

A while loop follows the following rule:

while (condition):

statement;

end

while and end are keywords for the while-loop statement. The condition defines the condition when one continues entering the while-loop. The statement defines the code to execute inside the loop.

### 3.5.3 The for Statement

A for loop follows the following rule:

for element:element\_data\_type in set:

statement;

end

for and in are keywords in the for-loop statement. The element denotes an element in the set, and after the execution of a for-loop statement, all elements in set should be iterated exactly once. The statement denotes the statement to execute when one enters the loop.

### 3.5.4 The run Statement

run is a keyword for dynamic geometric analytics. Each run must correspond to an instant of runset. Each geometric shape that will be dynamically analyzed in the run statement must be added into the runset instant at first. A typical run statement is shown as follows:

run runset\_name:

dynamic analytics sentences;

change parameters for the next time run;

end

## 3.6 STD Library Reference

This chapter introduces the STD library of Geo. STD library (std.glib) is a file preloaded by Geo compiler before compiling the programs. STD library includes system functions declaration, geometric types (including build-in functions) declaration.

Part A: System constant declaration.

PI = 3.14159;

Part B: System function declaration.

function print(info:int const, ...):void;

function print(info:float const, ...):void;

function print(info:char const, ...):void;

function print(info:bool const, ...):void;

function print(info:string const, ...):void;

function int\_to\_string(input:int const):string;

function float\_to\_string(input:float const):string;

function bool\_to\_string(input:bool const):string;

function char\_to\_string(input:char const):string;

Part C: Geometric types declaration.

//controltype runset

model runset:

runEnable:bool;

times\_of\_run:int;

shape[]:geometricShape;

runpara[]:char;

runset(times\_of\_run:int, g1:geometricShape, run\_para\_g1:char, ...);

function refresh():void;

function addElement():bool;

function removeElement(g:geometricShape,para:char):bool;

function enableRun():void;

function disableRun():void;

end

model geometricShape:

step[]:float;

stepSet[]:bool;

geometricShape();

end

//geometricShape submodel: dot

//dot parameter name: 'x' 'y'

model dot:

topmodel(geometricShape);

step[2]:float;

stepSet[2]:bool;

dot(x:float, y:float);

function getX():float;

function getY():float;

function distance(dot1:dot const):float;

function distance(line1:line const):float;

function setRunstep(val:float,pos:char):void;

function getRunstep(pos:char const):float;

end

//geometricShape submodel: line

//line parameter name: 'a' 'b'

//line formula y = ax + b

model line:

topmodel(geometricShape);

step[2]:float;

stepSet[2]:bool;

endPoint[2]:dot;

endPointset:bool;

a:float;

b:float;

line(a:float,b:float);

line(dot1:dot,dot2:dot);

line(a:float,b:float,endpointx1:float,endpointx2:float);

line(dot1:dot,dot2:dot,endpointx1:float,endpointx2:float);

function getPara(pos:char const):float;

function getY(x:float const):float; //Exception may occur.

//If endPointset = false, return [0,0]

function getMidpoint():dot;

function setRunstep(val:float,pos:char):void;

function getRunstep(pos:char char):float;

function intersect(polygon1:polygon const):dot[];

function intersect(circle1:circle const):dot[];

end

//geometricShape submodel: circle

//circle parameter name: 'a' 'b' 'r'

//circle formula r^2 = (x-a)^2 + (y-b)^2

model circle:

topmodel(geometricShape);

step[3]:float;

stepSet[3]:bool;

a:float;

b:float;

r:float;

circle(center:dot,radius:float);

function setRunstep(val:float,pos:char):void;

function getRunstep(pos:char):float;

function getCenter():dot;

function getRadius():float;

//Out\_of\_range Exception may occur.

function getY(x:float const):float[];

function intersect(polygon1:polygon const):dot[];

function intersect(circle1:circle const):dot[];

end

//geometricShape submodel: polygons

//polygons parameter name: 'a' 'b' ...

//polygons formula: A set of dots

model polygons:

topmodel(geometricShape);

step[]:float;

stepSet[]:bool;

apexs[]:dot;

polygons(num\_of\_apex:int,apex[]:dot);

function setRunstep(val:float,pos:char):void;

function getRunstep(pos:char):float;

function getCenter():dot;

function getRadius():float;

//Out\_of\_range Exception may occur.

function getY(x:float const):float[];

function intersect(polygon1:polygon):dot[];

end

# Chapter 4 - Project Plan

## 4.1 Project Management

### 4.1.1 Planning

Our group members met every Friday afternoon since we were all available at that time. The whole project was divided to 4 stages, proposal, language design, basic functions implementation and graph functions implementation. Initially we worked individually on small problems, but as the project progressed, we found it more efficient to work together on major challenges. We chose CLIC as our main workspace, most part of this project was finished there.

### 4.1.2 Specification

We intended to implement everything exactly as stated in the LRM, however, later we realized that it was impossible to stick to it since many problems were not considered in the LRM. So we decide to update the reference manual as we develop our language. For example, in the first version of LRM we required user to specify the data type of iterator and set in the for loop, but since the iterator must belong to set there was not need to specify its type. And some features in the first LRM are not implemented because they are too hard or not necessary. For example, we planned to let the panel capable of showing two images at one time, however, in python it was too hard to realize, so we had to skip this part.

### 4.1.3 Development

To coordinat works of different members and keep track of our progress, we use Git as our version control system with Github as the host. When members have completed and tested their works, they push them to the master branch of our Geo project. Then we will look over the codes together, make improvements and combine different parts. Most time we work together in CLIC, especially the final stage of the project, because we can discuss together if any problem occurs.

### 4.1.4 Testing

A first simple test case was written for the hello world demo. Then we added many individual test cases, each focusing on some small functions of the language. Since the project is quite big in size, we have to test it this way otherwise we may easily get confused about which part is wrong. Also each test case is prepared with an expected result, if the actual output of the source code is not the same as our expected output then the compiler needs correction of this function.

## 4.2 Programming Style Guide

 • Initial lower case for function and variable names.

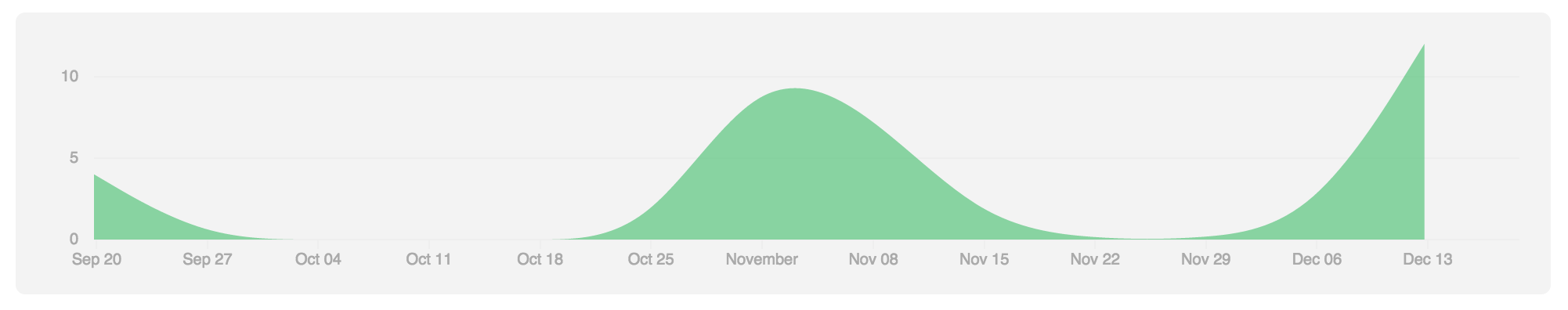
• Codes after the “let” statement and pattern matching are indented.

• Avoid breaking expressions over multiple lines.

• Commonly reused code are written as a helper function.

## 4.3 Project Timeline

The following graph shows our commits to Github during the whole semester.



Sep 28, 2015

Finished proposal

Oct 15, 2015

Started with LRM

Oct 25, 2015

Finished LRM

Oct 30, 2015

Started with scanner and parser, AST

Nov 02, 2015

Finished scanner, parser and AST, started with compiler

Nov 07, 2015

Continued on compiler, modified parser and AST

Nov 07, 2015

Finished compiler, prepared simple test cases

Nov 16, 2015

Hello world demo

Nov 20, 2015

Started with semantic check, code generation and testing

Nov 30, 2015

Continued on semantic check, code generation, and testing

Dec 06, 2015

Updated SAST, code generator, added test cases

Dec 16, 2015

Finished with SAST and code generator, added more test cases

Dec 17, 2015

Started with final report, prepared for demo

Dec 20, 2015

Finished final report, demo worked well

Dec 21, 2015

Final presentation

## 4.4 Roles and Responsibilities

Zichen Chao - System Architect (worked on paser, ast, compiler, the major contributor of sast)

Qi Wang – Product Manager (worked on parser, ast, copiler, wrote final report and slides)

Yuechen Zhao - Language Guru (designed language features and helped with python functions)

Ziyi Luo – Tester (worked on scanner, created test suites and Geo python libraries)

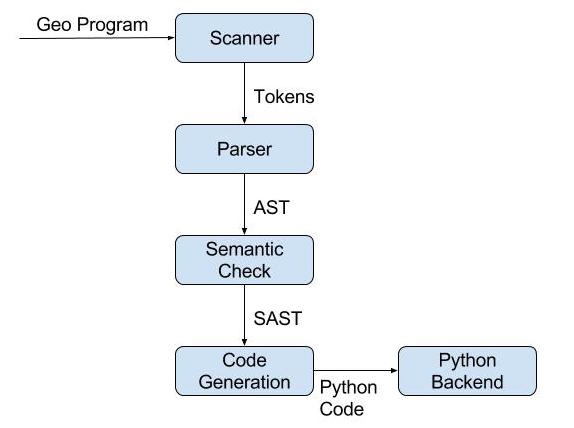
## 4.5 Software Development Environment

The Geo project was built on OS X and as stated above, Git was used as a distributed version control system and Github was the host. We used sublime to write the compiler, including the scanner, parser, ast, sast, compiler and test cases. The language we used was mainly OCaml, and python for augmenting the graph visualization and animation. Lastly, makefiles were used to automate the process of compiling, shell scripts were used for testing.

## 4.6 Project Log

# Chapter 5 - Architectural Design

## 5.1 Block Diagram



## 5.2 Components

### 5.2.1 Scanner

The scanner was implemented with ocamellex, with scanner.mll as the corresponding file. The scanner breaks down formatted input into tokens and translates individual tokens to some basic data types like int and char, discarding unnecessary information for compilation like whitespaces and comments. Also, it does basic syntax checking, any illegal symbols like # or $ are discovered and rejected.

### 5.2.2 Parser and AST

We use ocamlyacc to implement the parser and ast, with parser.mly and ast.ml as the corresponding files. The parser parses the token stream produced by the scanner to an abstract syntax tree, which represents the abstract syntactic structure of the source code. The parsing process does further check of syntax following the specifications in ast file.

### 5.2.3 Semantic Check and SAST

The semantic check was implemented using OCaml, and the associated file is compile\_sc\_py.ml, which takes in an abstract syntax tree and produces a semantically checked tree of python. A standard python AST, pyast.ml, is used for comparing. Errors including inconsistent types in expressions, reference to undefined functions or variables are discovered during this process. We use global and local maps to record functions and variables in the scope and raise undefined exceptions if there is no match in maps.

### 5.2.4 Code Generator

The generator was implemented using OCaml and the associated file is compile\_to\_pycode.ml. The generator takes in the sast produced before and generates python code from it. Most of the code it generates is hard coded by translating python.

# Chapter 6 - Test Plan

# Chapter 7 - Lessons Learned

## 7.1 Qi Wang

I have learned how to do a team project. When a project is extremely large and complex, like this one, it’s impossible to finish all the works individually. So communications are really important, we need to keep communicating with others about our own sub-tasks, and we should be able to combine different parts together. This is not easy, since many details are involved. And plan is very important, the first thing before start is to make a plan and follow it strictly. Even if we have a perfect plan, we still need to keep communicate with our teammates during the group project. Sometimes things can be different from what we have imaged. For example, the semantic check is far more difficult than we thought, so as soon as I found out that it might take us much more time, I told my teammates and we discussed on how to change the schedule.

## 7.2 Zichen Chao

The most important thing I have learned from this project is that we need to keep the whole picture in mind. There are many technologies, tools, theories and other details involved in making a compiler. In each part of the project, we have to apply several of these in our codes or designs. When we are doing the coding or design, these sub-tasks seem to be independent with each other. For example, the parsing tree of the code and the semantic check of the code seem to be two very different tasks. It is possible that we may not see any connection between these two things at the first sight. However, when we were implementing the semantic check, we found that several modifications on the parsing tree had to be made to help with the semantic check.

## 7.3 Yuechen Zhao

If I can use only one word to describe the key to success in this project, I will say communication. This is a large project and we have 4 members in our team. When working together, we have to go through many discussions because different people have different opinions. We need to make an agreement on one plan before we can start to divide the work, and it often takes us a lot of time to discuss the advantages and disadvantages of ideas. But the discussions do help us to save time from trying things that not work for us. During the discussion, we should be able to explain our opinions clearly and listen carefully to others. All these make our communications efficient.

## 7.4 Ziyi Luo

Before this course project, I have never realized that how important testing is in a project. After finishing the most part of our project, we decided to try several test cases. I thought our compiler was able to handle all the test cases, or we could make some little modifications to correct it.

However, when the test was done, we discovered so many problems and we were not able to resolve them by a few lines of codes. We had to adjusts the foundation part of our complier and rewrite several parts. We realized that we should spend more time on testing before finishing the compiler, since it’s much harder to resolve all the bugs in the end. So we designed many test cases, each one testing a single function of our compiler, to resolve problems one by one.

# Chapter 8 – Appendix