Geo

A geometric solution language

Language Reference Manual

ver 1.1

Group Members

PM (make sure conductible, write-ups):

Qi Wang (qw2197)

Language Guru (design language syntax):

Yuechen Zhao (yz2877)

System Architect (based on syntax, build compiler):

Zichen Chao (zc2321)

Tester (test test test test):

Ziyi Luo (zl2471)

(COMS W4115) Programming Languages and Translators

Fall 2015

Nov. 7, 2015

Contents

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.

Chapter 2 - Lexical Elements

This chapter specifies the lexical elements of Geo programming language.
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.

2.1 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
```

2.2 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
```

2.2 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
```

2.3 Constants

2.3.1 Integer Constants

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

```
0 1 20 +210 -15
```

2.3.2 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
```

2.3.3 String Constants:

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

```
"Hello world!" "1234567890" "Name\tID\tScore\n"
```

2.3.4 Mathematical Constants

Mathematical constants include some frequently used parameters such as:

```
PI = 3.14159
```

2.4 Separators

Several characters are used as the separators:

```
( ) { } [ ] ; , .
```

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

2.5 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:

```
.
* / % ^
+ -
> >= < <=
!
& |
== !=
=
```

Chapter 3 - Data Types

This chapter introduces all standard data types in Geo.

3.1 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 Geometric Types

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

3.2.1 `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 `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.

3.2.3 `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.

3.2.4 `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 `dot`s. 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.

3.2.5 `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.

3.3 `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);
```

3.4 `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.

Chapter 4 - Expressions

4.1 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.

4.1.1 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;
```

4.1.2 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;
```

4.2 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.

Chapter 5 - 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.

5.1 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[]`.

5.2 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.

5.3 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, ...);
```

Chapter 6 - Compound Statements

6.1 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.

6.2 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.

6.3 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.

6.4 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
```

Chapter 7 - 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 8 - Typical Program Structure and Examples

In this chapter, a typical Geo program is introduced for better understanding of Geo.

8.1 Typical Geo Program Structure

A typical Geo program includes:

1. panel presets;
2. `function` declaration and definition;
3. geometric shape declaration and initialization;
4. `runset` declaration and initialization;
5. `run` statement description.

8.2 Geo program example

An example followed the structure description in Section 8.1 will be shown as below.

```
//panel presets
@panel panel_demo
@mode figure

//function declaration and definition
function print_dot_list(dots[]:dot):void:
    for d:dot 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
```

-- End of GEO LRM --