Language Specification

GeoX

Compilers

భారతీయ సాంకేతిక విజ్ఞాన సంస్థ హైదరాబాద్
भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

TEAM MEMBERS:

Full Name	**Roles**
G. Chandrahas	Project Manager
D. Hemanadh	Tester
K. Prudhvi	System Integrator
K. Kedareswar	Language Guru
A. Deddeepya	Tester
K. Karthik	System Architect
M. Manohar	System Integrator
B. Rutwik	System Architect

Instructor: Dr. Ramakrishna Upadrasta

Contents

Introduction

1.1 Background

GeoX is a domain-specific language (DSL) designed to simplify working with geometric shapes on a 2D coordinate plane. It specializes in analytical geometry, focusing on the properties, relationships, and characteristics of shapes like triangles, circles, parabolas, ellipses, and more. With GeoX, users can easily define geometric shapes using coordinates and equations, then perform operations such as calculating areas, distances, and angles, or applying transformations.Our DSL is case-sensitive.

GeoX was developed to address the complexity of geometric computations in general purpose languages. Its simple syntax and specialized functions make tasks like calculating intersections and distances more accessible. GeoX aims to enhance learning and support advanced applications in fields like computer graphics, robotics, and engineering. It achieves this by reducing complexity and improving the efficiency of geometric analysis.

1.2 Aim

The goal of GeoX is to provide a simple and effective tool for working with geometry. It aims to make geometric tasks faster and easier for both learning and professional use, helping users understand concepts better and save time in calculations and analysis.We want to simplify the amount of work needed to perform calculations on 2D shapes so the problem solvers can worry more about coming up with the algorithm to solve the problem rather than get stuck with thinking about writing long complex functions to solve even basic geometry sums.

Data Types

2.1 Primitive Datatypes

DataType	Initialization	Description
void	`<void> funct ();`	Represents the absence of a value.
string	`string a = "GeoX";`	Sequence of characters (length > 1).
bool	`bool a = false;`	Boolean value (true or false).

Table 1: Primitive Datatypes in GeoX

2.2 Non-Primitive Datatypes

Data Type	Initialization	Description
Num	`num n = 2;`	Represents any type of number.
Point	`point p(1.2, 2.5);`	Represents a point with two floats (x, y).
Equation	`equation e("y = mx + c");`	Represents an equation as a string.
Line	`line l(p, 5.6);`	Represents a line with a point and a slope.
	`line l(p, p);`	Represents a line with two points.
	`line l(e);`	Represents a line with equation.
Circle	`circle c(p, 8.1);`	Represents a circle with a center point and radius.
	`circle c(e);`	Represents a circle with equation.
Parabola	`parabola q(p1, p2);`	Represents a parabola with a focus and vertex.
	`parabola q(e);`	Represents a parabola with equation.
Hyperbola	`hyperbola h(p, 4, 10);`	Represents a hyperbola with center, a, and b.
	`hyperbola h(e);`	Represents a hyperbola with equation.
Ellipse	`ellipse e(p, 1, 4);`	Represents an ellipse with center, a, and b.
	`ellipse e(e);`	Represents a ellipse with equation.

Table 2: Non-Primitive Datatypes in GeoX

Lexical Conventions

3.1 Key Words

All the datatypes are included in key words.

if	elif	el	loop
stop	skip	return	switch
goto	write	read	void
bool	struct	include	define
null	main	default	case
true	false		

Table 3: Key words in GeoX

3.2 Comments

The basic syntax of **GeoX comments** includes:

- **Single line comment** - Uses '##' for single-line comments.
- **Multi line comment** - Uses '#* ... *#' for multi-line comments.

This is the basic syntax of **GeoX Comments**:

```
Single-line comment - ##
## This line has been commented.

Multi-Line Comment - #* ... *#
#*
    This is a multi-line comment
    This is a multi-line comment
*#
```

3.3 White Spaces

White spaces, new lines and tabs are ignored.

3.4 Identifiers

An identifier is a name used in a program to represent a variable, function, or any other user-defined entity.

- Variable starts only with uppercase or lowercase character.
- Variables are case-sensitive and don't contain white spaces, # and punctuations.
- Keywords and datatypes can't be used as variables.

This is the basic syntax of **GeoX Identifiers**:

- Correct Declaration:

```
## variable_names
## var
## Var
## var123
## Var_12
```

- Incorrect Declaration:

```
## 12var
## _Var
```

3.5 Punctuators

Name	Symbol
Colon	:
Semi colon	;
Comma	,
Single quote	’
Double quote	”
Flower Brace	{}
Square Bracket	[]
Paranthesis	()
Angular Bracket	<>
Dollar	$

Table 4: Punctuators in GeoX

3.6 Special Characters

Character	Symbol
Tab Space	\t
New line	\n

Table 5: Special Characters in GeoX

Operators

Operators	Description	Associativity
++, - -	Postfix inc && dec Unary Operators	Left to right
++, - -	Prefix inc && dec Unary Operators	Right to left
!	Logical Not	Right to left
^	Exponentiation	Right to left
*, /, %	Arithmetic	Left to right
+, -	Arithmetic	Left to right
==, !=, >=, <=, <, >	Relational Operators	Left to right
&&, ||	Logical operations	Left to right
=	Assignment	Right to left
~	Used for seperating print statements	Left to right

Table 6: Operators in GeoX and Their Associativity

Declarations

5.1 Variable Declarations

- A variable declaration gives a name to a memory in the program where data is stored. Here are some examples of variable declarations in our DSL.

This is the basic syntax of **GeoX Variable Declarations**:

```
num radius = 5;
string s = "GeoX";
point p(2, 5);
circle c(p, radius);
eqaution eq = "x^2 + y^2 = 1";
```

5.2 Function Declarations

- A function is a reusable block of code designed to carry out a particular task. It can be invoked with input values (parameters) and perform operations specified.

This is the basic syntax of **GeoX Function Declarations**:

```
<data_type> function_name ( data_type arguments ) {
    ## block
}
```

Program Flow

GeoScript programs are composed of the following key components:

- **Headers and Declarations:** Specify any necessary library imports or global settings.

```
$include "geo.core";  ## Import core geometric functionalities
```

- **Define Macros:**

```
$define Pi 3.14;  ##Precision for calculations
$define MAX_SHAPES 12;
```

- **Variable Declaration:**

```
num shape_counter = 0;
num total_area = 0.0;
bool is_active = true;
point origin(0, 0);
circle C1(origin, 5);
```

- **Function definitions:** Define functions that perform specific geometric computations or transformations.
- **Main Program:** The core logic, including loops, conditionals, and operations on geometric shapes.
- **Example code:**

```
## Headers and Declarations
## Import core geometric functionalities
$include "geo.core";
## Import utility functions for geometry
$include "geo.utils";

## Define Macros
$define PI = 3.14159;  ## Define the value of Pi
## Define a macro for squaring a number
$define N 1;

## Shape Definitions
point A(2, 3);  ## Define a point A at coordinates (2, 3)
point B (5, 7);  ## Define a point B at coordinates (5, 7)
line l1(A, B);  ## Define a line AB between points A and B

## Define a circle C with center A and radius 4
circle c1(A, 4);
#*
Define an ellipse with center A, major axis 3
```

```
and minor axis 5
*#
ellipse elli(A, 3, 5);

## Functions
<num> distance(point p1, point p2) {
    ## Calculate the Euclidean distance between two points
    return sqrt((p2.x - p1.x)^2 + (p2.y - p1.y)^2);
}

<num> area(circle c) {
    ## Calculate the area of a circle
    return PI * SQR(c.radius);
}

## Main Program
<void> main () {
    ## Calculate the distance between A and B
    num d = distance(A, B);
    ## Calculate the area of circle C
    num area_C = area(C);
    ## Output the distance between A and B
    write ~ "Distance between A and B:" ~ d;
    ## Output the area of circle C
    write ~ "Area of Circle C:" ~ area_C;
}
```

Statements

GeoX Statements are commands in a program that control the flow of programme, such as assigning values, repeating tasks, making decisions or running functions.

7.1 Labelled statements

Labelled statements are used to change the flow of a program to the given statement from a specific position. The identifier followed by " : ".
This is the basic syntax of **GeoX Labelled statements**:

```
label: {
    ## Code block
}
goto label: {
    ## goes to the code block of label
}
```

7.2 Expression statements

Operators combined with constants or variables form an expression. Expression can be evaluated to a value. Expression statements are ended with ' ; '.
This is the basic syntax of **GeoX Expression statements**:

```
num a = 10;
point p(1, 2);
num b = a*10 + 20;  ## Expression statement
p.x = p.x * p.y;  ## Expression statement
```

7.3 Declarative statements

In a program, declarative statements declare a name and data type.
This is the basic syntax of **GeoX Declarative statements**:

```
num a = 10;  ## Declarative statement
point p(3, 4);  ## Declarative statement
circle c(p, 8); ## Declarative statement
```

7.4 Compound statements

A group of statements enclosed in curly braces { } is a compound statement.
This is the basic syntax of **GeoX Compound statements**:

```
if(a > 10) {
        ## Compound statement
}
elif(a <= 10) {
        ## Compound statement
}
```

7.5 Iteration statements

The statements enclosed in loop conditions are iteration statements. They are while loop and for loop.
This is the basic syntax of **GeoX Iterative statements**:

```
loop(num a = 0; a < 10; a++) {
        ## Iteration statement
 }
```

7.6 Jump statements

Jump statements enable the program flow by jumping to another specific code section, continue is used to skip rest of the statements in a loop or break is used to stop the program.
This is the basic syntax of **GeoX Jump statements**:

```
num a = 5;
loop(a < 10) {
        if(a != 7) {
        skip; ## skips to the next iteration
    }
    elif(a == 7) {
        stop;  ## stops the loop
    }
}
```

7.7 Selection statements

Selection statements select one of the several possible control flows. Program code block gets selected based on the specific condition.
This is the basic syntax of **GeoX Selection statements**:

```
num score = 5;
switch(score) {
    case 0: {
        ## Code block
        stop;
    }
    case 1: {
        ## Code block
        stop;
    }
    default: {
        ## Code block
    }
}
```

Possible Optimizations

Here are some of the ideas we had to optimize programs in our DSL:

- If the user want to calculate intersection points of two circles, first we check the distance between two centers and if that distance $d > r_1 + r_2$, then we return no points.
- If we have two points on a curve and we need to determine the angle between tangents to the curve at these two points, we check if there is a focal chord between those two points, if yes, then angle is 90 and the point of intersection lies on directrix of that curve.
- Given a circle equation and a point on it, if we need to calculate angle between tangent to circle from that point to any line, we check if the line passes through the center of circle and that point, if yes then the line is normal and angle is 90.
- Distance between P and point for a given line and point. P is the image of point w.r.t line. Distance between P and given point = 2 * d (d is distance between point and line).
- Similar to above point if line is directrix (we don't know whether the line is directrix or not) and point is focus so distance is 2 * a for parabola.
- Instead of using the full equation of an ellipse to verify if a point lies on the ellipse, you can use the property, the sum of distances from the point to the two foci is 2a or not. If it is 2a then point lies on the ellipse. If it is less than 2a then point lies inside the ellipse and if it is more than 2a then point lies outside the ellipse.
- Reflecting a curve across a line y = x can be done by swapping x and y in the curve equation.
- Distance between p1 and p1 is 0.

We are planning to implement many more optimizations like these.

Example Program

```
## Import headers
$include "geo.core";
$include "geo.utils";

<void> main( ) {
    ## Define points
    point p1(1, 2);
    point p2(3, 4);

    ## Define a line between points
    line l1(p1, p2);

    ## Prints line equation
    write ~ "line equation : " ~ l1;

    ## Define a circle with center and radius
    circle c1(p1, 5);
    write ~ "circle equation : \t" ~ c1.equation;

    ## Calculate the tangent to the circle at a given point
    line tangent = c1.tangent(p2);
    num degree = angle_between(l1,tangent);

    ## Output results
    write ~ "Tangent line: " ~ tangent;
    write ~ "angle between tangent and l1 : " ~ degree ~ \n;
}
```