A Language for Geometry Reference Manual

Mengqi Zhang, Shiyao Zhu, Ainur Rysbekova, Qiang Deng, Qingye Jiang
Department of Computer Science
Columbia University in the City of New York
New York, NY 10027 USA
{mz2369,sz2395, ar3005,qd2114, qj2116}@columbia.edu

October 29, 2012

Contents

1 Introduction

A Language for Geometry (ALG) is a programming language for geometry calculation. With ALG, user can calculate certain attributes of geometric figures and relationship between them in a convenient way. Also user can verify his/her speculations on geometry rules and witness how the other attributes will change as certain conditions vary.

ALG programs are first compiled into "C" modules and will be further compiled into machine binary by C Language compiler.

2 Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. Space, tabs and newlines are symbols of separating tokens. Every token should match the longest string that could comprise a token in the input buffer. For example, if there following two characters in the input buffer are "∼ " and "=", we will get the token from "∼= " rather than from "∼".

2.1 Comments

Symbol !! is used to mark a single-line comment.As for multi-line comments, (! marks the beginning of the comments, and !) means the end.

Here's an example:
! ! This is a single-line comment.
(! This Is a multi-line comment !)

2.2 Identifiers (Names)

An identifier is a sequence of and only of letters, digits, and underscores (_). An identifier must begin with a letter. ALG is a case sensitive language, so uppercase and lowercase are different for the compiler.

2.3 Keywords

The following terms are reserved as keywords for ALG:

int	point	for
float	line	while
string	polygon	if
array	ellipse	else
boolean		elseif
void		case
		return
		switch
		continue
		break
		def
		default

2.4 Constants

ALG has constants including Integer, float ,string, boolean and graph constants.

2.4.1 Integer constants

An integer constant is a sequence of digits without a decimal point.

For example:
int x = 15;

2.4.2 Float constants

A floating constant consists of an integer part, a decimal point, 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. Every floating constant is taken to be double precision.

For instance, float number can be like:
0.125
1.25
123.33542000

While these numbers are illegal:
1..2 (More than one decimal)
.123 (integer part missing)

2.4.3 String constants

A string is a sequence of characters surrounded by double quotes “ ¨ ” .

¨ This is a string¨ defines a string in ALG.

Two double quotes must use together, none of them could be missing.

2.4.4 Boolean constants

Boolean constants represent true or false.

Example:
boolean x = True;
boolean y = False;

Notice that True, TRUE and true all work for a true value. False, false and FALSE could all be used to represent false value.

2.5 Graph constants

We use a new section to introduce graph constants because of their importance to our language and the unique features.

2.5.1 point

Point is defined by a point [x, y] (x,y are floats)in the coordinate system. It is used to set a point.

Usage:
point a = [1,2];

2.5.2 line

Line is settled by two points. The two points should be on the line in order to define a line.

Usage:
line l1 = [[0,1],[1,2]];

2.5.3 ellipse

Ellipse is defined by a central point (not focus point) and two distance –a (the length of major semi-axex) and b (the length of minor semi-axes), and in the format as [point, a, b] (a and b are floats).

Particularly, when a equals b, the ellipse is a circle.

Usage:
ellipse ell = [[1,4] 3, 5];
ellipse circleOne = [[0,0] 5, 5];

2.5.4 polygon

For polygon, points on each edge are used to define it. In order to clarify ambiguities, the points must be introduced clockwise. The polygon is defined as the format [point, point,point,...,point]

Usage:
polygon triangle = [[0,0.1],[0,1.2],[1.4,1]];
polygon square = [[0,0],[0,1],[1,1],[1,0]];

2.5.5 Array Constants

We use array constants to initialize an array, but its optional. The format of an array constan is as {contant1,constant2,constant3,...,constantn}, all constanti (i=1...n) should be of the same data type.

2.5.6 Explanation of constants of Array and Graphs

The Graph Constant and Array Constant will not be recognized during the lexical analysis phase, but it will be recognized during the parser phase.

2.6 void

void is a special data type that neither belongs to any other data types nor has any concrete meaning. It's mainly used to maintain the whole format of the program and can be treated as "empty".

3 Operator Characteristics

We have defined many operators to deal with Geometry Figures(//,∼=,∧ , ⊢), as well as operators deal with both Geometry Figures and Algebraic Calculations (==, ! =, <<, >>, <, >). These operators' operands cover a large range of objects. For example, "∼=" could be applied to two operands of type ellipses or Polygons, and "∧ " could be applied to two operands of type (line, line),(ellipse, line), (ellipse, ellipse), etc. The details will be discussed in the section 7 of "Expression".

4 Type-Conversion and Type-Detection

There are many situations where the type of a integrity in the program need some modification. In general, there are two cases.

Case one: Some operators (binary) require the two operands should be of the same type; or a certain operator requires the operand(s) to be a certain type. In this case, the program should converted the operands (or one of them) to the appropriate types. We introduce Type-Conversion to solve this case.

Case two: Sometimes, which type one certain format in our language belongs to is ambiguous; it should be detected with the circumstances. We introduce Type-Detection to solve this case.

Data Types :
Basic: int, float, string, boolean, array
Compound: point, line, polygon, ellipse

Example :
Type-Conversion: a=1.2+1; 1 should be converted to a float;
Type-Detection: line // [[1,2],[3,4]], the operand on the right should be interpreted as a line, rather than an array of two point because we use {} to indicate an array and [] to indicate the graphs.

4.1 Type-Conversion

a) Int and Float: int will be converted to float without loss of accuracy. When converted from float to int, just convert the integer part of the float to int; if the integer part is larger than the MAX_INT, then do truncation, only retain the least significant bits. Although float to int conversion is a little meaningless, this should be defined to prevent errors that may results from undefined behaviors.

i. Int_To_Float Conversion:
  int a=1;
  float b=(float) a;
  outcome: b=1.0
ii. Float_To_Int Conversion:

```
float b=1.2;
int a=(int)b;
outcome: b=1
```

b) Boolean and Int: When boolean turns to int, "true" turns to "1" and "false" turns to "0". When int turns to boolean, positive ones turn to "true", and zero or negative ones turn to "false";

i. Int_To_Boolean Conversion:

```
int a=2;
boolean b=(boolean) a;
outcome: b=true;
int a=-1;
boolean b=(boolean) a;
outcome: b=false;
```

ii. Boolean_To_Int Conversion:

```
boolean a=true;
int b=(int) a;
outcome: b=1;
boolean a=false;
int b=(int) a;
outcome: b=0;
```

c) Boolean and Float: When boolean is converted to float, "true" turns to float "1.0" and "false" turns to float "0.0". When float turns to boolean, positive float numbers turn to "true", and zero and negative ones turn to "false".

i. Float_To_Boolean Conversion:

```
float a=1.5;
boolean b=(boolean)a;
outcome: b=true;
float a=-1.2;
boolean b=(boolean)a;
outcome: b=false;
```

ii. Boolean_To_Float Conversion::

```
boolean a=true;
float b=(float) a;
outcome: b=1.0;
boolean a=false;
float b=(float)a;
outcome: b=0.0;
```

d) Implicit Type-Conversion: Our language have policies to implicitly convert one type of operand to another type, according to the expressions, the policies are as follows:

Some statements requires certain types of parameters, in this situation, the compiler should automatically use the Type-Conversion rules above to convert the parameters to certain groups when they are not:

Example:

```
float a=1.5;
while(a)
{
```

}
If(a)
{

}

In this case, the variable a should be converted to boolean type according the the rules of Boolean and Float Conversion in c). The same when a is of type int.

When computations should be done between different types of values with the operator requiring two values of the same type, there are following policies to be followed:

int→float;

That is, when two operands of different types are applied to algebraic operators (+,-,*,/), the operand which with the type left to that of another operand in the above rule should be converted to the other type.

Example:
Int a=1;
float b=1.5;
float c=a+b;

The above operand a will first be converted to type float, and then do the addition.

e) Note: any conversions which are not mentioned above are illegal:

Example:
int a=1;
string str=abc;
a=(int)str;
The last statement above is illegal.

5 Expression

5.1 Primary Expression

5.1.1 Identifiers

An identifier indicate a variable or a function, and the example is like: rectangle1, getArea.

5.1.2 Constant

Decimal integer, floating, boolean, and constants are all considered as primary expressions. The type of a constant is defined based on its form and value.

5.1.3 Graph-Constant

point, line, ellipse, polygon constants are considered as primary expressions. Each geometry shape has its own patterns, like ellipse is [[1,2],1,2]. We can get the type of a shape based on this pattern.

5.1.4 String

A string is a primary expression. It is an array of characters which carries literal information.

5.1.5 (expression)

A parenthesized expression is a primary expression. Its type and value are identical to those of the unadorned expression. The parenthesized expression is used to state the calculation priority clearly.

5.1.6 primary-expression (expression-list)

A primary expression followed by a list of expressions indicated the appearance of a function call, and it is a primary expression. The primary-expression indicates which function (of which object 's) is being adopted (the following expression list can be empty). Commas separate the expression-list in the parenthesis. The expression list includes all the parameters that are used in the calculation of the function. The returning type is decided by the function declaration.

When the function is called, a copy of all parameters is made, and the function may not change the real value of the actual parameters. However, the properties or values of an object can be changed by some built-in functions.

5.2 Geometry Operators

5.2.1 expression//expression

Both operands must be lines. The value of "l1 // l2" is true when l1 and l2 are parallel and false when two lines are not parallel.

5.2.2 expression $\wedge$ expression

Both operands must be of the same type including line, ellipse. The value of "s1 $\wedge$ s2" is the intersecting point of s1 and s2 when s1 and s2 are both lines, or void when they are parallel. The value of "s1 $\wedge$ s" is an array of points when one operand is a line and the other one is an ellipse, or void when they do not intersect. The value of "s1 $\wedge$ s2" is the tangent or secant line of s1 and s2 when they are both circles (special cases of ellipse), or void when they do not intersect. "$\wedge$ " operator does not support the calculation between a non-circle ellipse and a circle, or between two non-circle ellipses.

5.2.3 expression $\vdash$ expression

Two operands can be line or ellipse. The value of "s1 $\vdash$ s2" is 1 when two shapes are secant, or is 2 when they are tangent, or is 3 when they are separate.

5.2.4 expression == expression

5.2.5 expression ! = expression

5.2.6 expression < expression

5.2.7 expression > expression

5.2.8 expression <= expression

5.2.9 expression >= expression

5.2.10 expression << expression

5.2.11 expression >> expression

5.2.12 expression <<= expression

5.2.13 expression >>= expression

Both operands should both be figures. The operators ==(two figures are the congruent), ! = (two figures are not congruent), < (area is smaller than), > (area is greater than), <= (area is less than or equal to), >= (area is greater than or equal to), << (perimeter is smaller than), >> (perimeter is greater than), <<= (perimeter is smaller or equal to), >>= (perimeter is greater or equal to) both yield 0 when the specified relation is false and 1 when it is true.

The "==" , "! =", "<", ">", "<=" and ">=" operators can also be considered as Boolean operators to judge whether two integers or floating numbers fulfill the specified relations. "==" is "equal to". "! =" is "not equal to". "<" is "smaller than". ">" is "greater than". "<=" is "smaller or equal to". ">=" is "greater or equal to". In that case, both operands should be integers or floating numbers. If one expression is floating number and the other expression is integer, then the integer will be converted into a floating number and then the comparison will be done based on the converted value.

5.2.14 expression ∼= expression

Both operands must be polygons. The value of "p1 ∼= p2" is true when p1 and p2 are similar, and the value is false when p1 and p2 are not similar.

5.3 Additive operators

The additive operators + and are binary and group left-to-right.

5.3.1 expression + expression

The result is the sum of the operands. If both expressions are integers, then the result is an integer. If both expressions are float, then the result is a float. If one expression is float and the other one is integer, then the integer will be converted into a float and the result will be a float number. No other combinations are allowed.

5.3.2 expression - expression

The result is the difference of the operands. If both expressions are integers, then the result is an integer. If both expressions are float, then the result is a float. If on expression is float and the other one is integer, then the integer will be converted into a float and the result will be float. No other combinations are allowed.

5.4 Multiplicative operators

The multiplicative operators *, / and % are binary and group left-to-right.

5.4.1 expression * expression

The result is the product of two expressions. If both expressions are integers, the result is an integer. If both expressions are float, the result is float. If one expression is integer and the other one is float, then the integer is converted into float and the result is float. No other combination is allowed.

5.4.2 expression/expression

The result is the division result. The same type considerations as for multiplication.

5.4.3 expression % expression

The result is the remainder from the division of the first expression by the second. Both operands must be integers.

5.5 Assignment operators

5.5.1 lvalue = expression

The value of object referred by lvalue will be replaced by the value of the expression. The operands' types can be divided into two situations. First, the operands are both numbers including integer and float. Second, the operands are both shapes including point, line, polygon, and ellipse. In the second case, the types of both operands should be the same. Other combinations are not allowed. The expression's type will be converted to the type of the l-value if the operands' types are not the same in the first situation.

6 Built-in functions

There are 5 built-in functions in our language.

def type-specifier move(type-specifier a, float x, float y)

The move() function takes three arguments, the first argument is the figure to be moved, which is either polygon or ellipse. The second and third arguments are the move distances: x and y correspond to horizontal and vertical distances respectively. The return value is the same as the type of the first argument.

def void print (type-specifier a)

The print function prints the argument it takes to the standard output. It's return type is void.

def void display (type-specifier a)

The display function displays the figure of the argument to the standard output. The argument type is restricted to point, line, polygon and ellipse. The return type is void.

def float area (type-specifier a)

The area function calculates the area of its argument. The type-specifier of the argument is either polygon or ellipse. The function returns the area of argument in type float.

def float perimeter (type-specifier a)

The function perimeter calculates the perimeter of its argument. The type-specifier of the argument is either polygon or ellipse. The function returns the perimeter of the argument in type float.

Note: The entry function of our language should be named as “main”.

7 Declarations

The general format of a declaration is:

(def) type-specifier decl

The type-specifier declares type of the following decl and instructs compiler to interpret the decl. The identifier def only applies to define a function, but not for variable declarations.

7.1 Type-specifier

Type-specifiers in ALG have similar types with C as well as its own types. They are:

int	boolean
string	point
float	line
void	ellipse
array	polygon

7.2 Function Declaration

The type-specifier of a function specifies the type of return value from the function. The return type of the function can be each of the above types in 7.1. However, every function can only have one certain return value.
The decl part consists of a function name, arguments of the function and body of the function. A specific format of a function declaration is:

def type-specifier fname (farg-list) fbody

fname is an identifier for the function. farg-list contains at least one argument, and each argument has its own type-specifier and its name, different arguments are separated by “,”. fbody implements the function, which consists of statements and variable declarations(see below).

A common example of function declaration is:

def float AreaDifference (polygon a1, polygon a2){
return (area(polygon a1) area(polygon a2)) }

7.3 Variable Declaration

Each variable has its own type as its type-specifier indicates. The decl part the variable declaration may contain one or more identifiers, while each identifier can be a specific variable or a variable array, separated by “,”.

Here is the example of the format:
type-specifier identifier1, identifier2, identifier3;

Identifiers following the same type-specifier have the same type, and either all of them are specific variables or all of them are variable arrays. Also, identifiers can be assigned specific values by “=” when declaring variables, but this assignment is optional.
A specific variable is identified with a name, while a variable array is identified with the format of array-name[counter]. The counter must be of type int. It specifies number of variables in the same array and stores the variables from name[0] to name[counter-1].

Common examples of variable declaration are:

int a = 2;
int a[int n]; (! value assignment is optional when declaring variables !)
polygon triangle1 = [point a1, point a2, point a3], triangle2 = [point b1, point b2, point b3];
polygon triangles[2] = [point a1, point a2, point a3], [point b1, point b2, point b3];

8 Statement

Note: $<$ *statement* $>$ is a general denotation of all kinds of statements in the following parts.

8.1 Expression statement

expression;

An expression statement is an expression followed by a semicolon. Operations like assigning value or calling functions are usually in this form. Semicolon after $<$ *statement* $>$ indicates the end of a statement.

Example:
a=3;
add(1, 2); !!add is a function with declaration int add(int, int);

8.2 Block statement

$$\begin{array}{ll} \{ & < statement1 > \\ & < statement2 > \\ & < statement3 > \\ & \vdots \\ \} & \end{array}$$

Block statement includes one or more statements and always enclosed on both sides by curly brackets. Block statement is treated as one single statement.

8.3 Conditional statements

if (expression)

    $< block\ statement >$

if (expression)
    $< block\ statement1 >$
else
    $< block\ statement2 >$

if (expression1)
    $< block\ statement1 >$
elseif (expression2)
    $< block\ statement2 >$

Conditional statements are similar to C's conditional statements.

In the first schema if expression returns a true value, the $< block\ statement >$ is executed. Otherwise the block statement is ignored.

In the second schema if expression returns a true value the $< block\ statement1 >$ is executed. Otherwise the $< block\ statement2 >$ is executed.

In the third schema if expression1 returns a true value the $< block\ statement1 >$ is executed. Otherwise expression2 is evaluated and if it yields a true value the $< block\ statement2 >$ is executed.

8.4 Repetitive statements

while (expression)
    $< block\ statement >$

for (expression1; expression2; expression3)
    $< block\ statement >$

While statement runs as long as expression holds true.

For statement repeats until expression2 is true. expression1 holds initialization, expression2 contains condition and expression3 executes increment after each iteration.

8.5 Break statement

break;

Break statement ends the execution of a repetitive statement or a selection statement.

8.5.1 Continue Statement

continue;

Continue statement ends this round of iteration and continue executing the following rounds of iterations.

8.5.2 Selection statement

switch (expression)
    $< block\ statement >$

Block statement within switch statement contains case labels, default statement and break statement:

case expression1:
    $< statement1 >$
    break;
case expression2:
    $< statement2 >$
    break;
    $\vdots$
    default:
    $< statement >$

If one of the case expressions matches the value of the switch expression, the $< statement >$ following that case expression is executed. If no case expression is equal to the value of the switch expression then $< statement >$ following default statement is performed.

8.6 Return statement

return expression;

Return statement indicates an exit from the current function and returns a value to the calling function. The returned value is an expression following the return statement.

9 External Definitions

ALG users can declare new variables, arrays and functions by composing a sequence of external definitions.

9.1 External functions definitions

A function is defined in the following way:

function-definition:
def type-specifier fname (farg) {fbody}

fname is the name of this function. farg is a list of all the parameters that the function will use. In the farg, each parameter's type should be given.

fbody is a compound statement which defines the detailed definition of the function.

10 Scoping Rules

An identifier (variable or function) is only visible after it is declared. This can be called a global scope. If an identifier is called before it is declared, then it will be regarded illegal.

For Example:
ellipse E1 = [[1.5,2.0],3,4]; !! this is legal
line l1 = E1 $\wedge$ E2; !! this is illegal because E2 has not been declared
E2 = [[1,2],1.5,3.2];