Geova

Language Design and Example Programs

Version 1.0.0

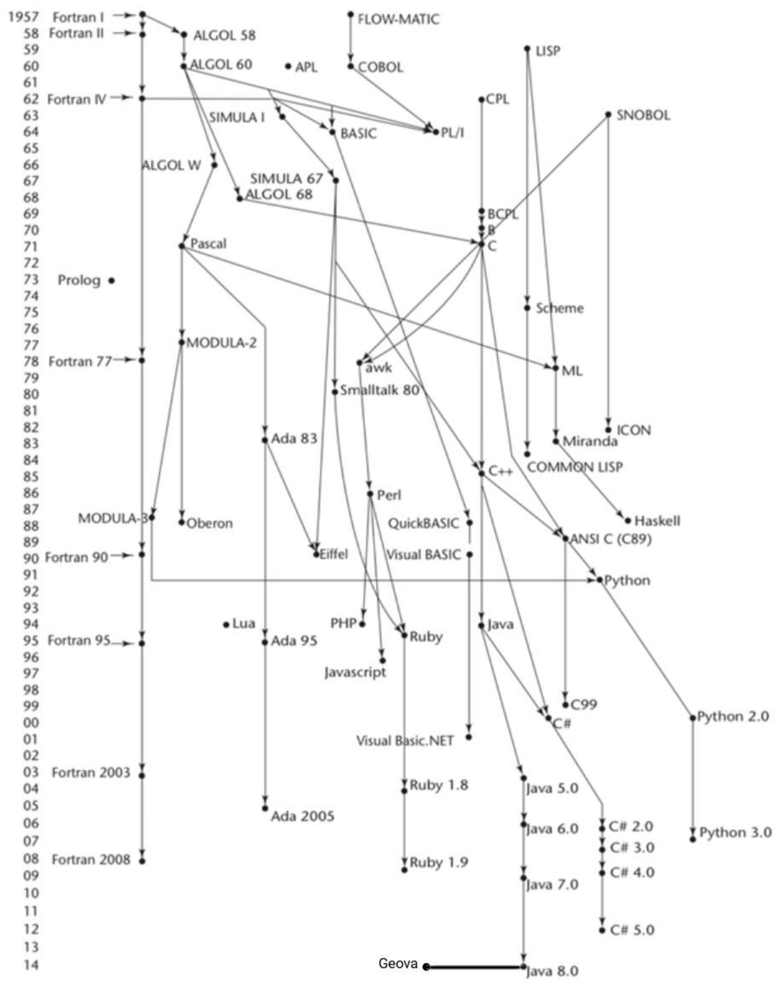
Anthony Sasso

1.Introduction

Geova is a simple, modern, object-oriented, and (strongly) type-safe programming language. It is based on Java and geographic/geopolitical history, and differs from java in the following ways:

1. Brackets have been removed and replaced with *found* and *dissolve*.
2. There are no classes in Geova, only worlds.
3. There are no Strings in Geova, only Countries.
4. There are no integers, floats, or doubles in Geova, only coordinates.
5. Functions in the Country world have been termed according to Geova's theme.
6. Variable assignment is accomplished using the walrus operator (:=).
7. Statement.broadcast() is System.out.println().

1.1.Genealogy



1.2.Hello world

```
public void helloWorld()

found

    Statement.broadcast("Hello World!");

dissolve
```

1.3.Program structure

The key organizational concepts in Geova are as follows:

1. Worlds (classes) are required to be in a universe (package). Universes can contain any number of worlds as long as they have unique names.
2. Worlds can have any number of custom attributes and functions, accessed using World.function() or World.attribute.
3. Functions are declared with explicit return and parameter types. Parameters are matched by location.
4. Universes, worlds, functions, repetition statements, and alternation statements must begin and end with *found* and *dissolve*.
5. A main method is required.

This example:

```
universe Reality
found
    public world Earth
    found
        private Country largest;
        private Country smallest;
        public Earth(Country large, Country small)
        found
            this.largest := large;
            this.smallest := small;
        dissolve
        public Country getLargest()
        found
            return largest;
        dissolve
        public void setLargest(Country large)
        found
            this.largest := large;
        dissolve
        public Country getSmallest()
        found
```

```
            return smallest;
        dissolve
        public void setSmallest(Country small)
        found
            this.smallest := small;
        dissolve
    dissolve
dissolve
```

declares a world named `Earth` in a universe called `Reality`. The `Earth` world contains two attributes: a Country named `largest` and a Country named `smallest`. The default constructor takes two Countries as arguments and matches them positionally to the actual parameters. Getter and setter methods have been created for both attributes in this world to allow them to remain private. Creating an instance of the `Earth` world would be done with the command:

```
private Earth ourEarth := new Earth("Russia", "Vatican City");
```

This example shows ourEarth being created with the Country "Russia" being assigned to the variable `largest` and the Country "Vatican City" being assigned to the variable `smallest`.

1.4.Types and Variables

There are two kinds of types in Geova: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. See Section 3 for details.

1.5.Visibility

In Geova visibility is defined as either public or private. Anything declared with visibility private will only be visible inside the same block in which it is defined. Blocks begin with the keyword *found* and end with the keyword *dissolve*. Anything declared with visibility public is visible to the entire world. Anything declared without visibility is treated as local and will not extend outside of the declaration block.

1.6.Statements Differing from Java

Statement	**Example**

Assignment statement	`this.largest := large;`
	`this.smallest := small.`
	
	`coordinate location := 432;`

If statement	`if (9 > 7)`
	`found`
	`    Statement.broadcast("This will output");`
	`dissolve`
	`else`
	`found`
	`    Statement.broadcast("This will not");`
	`dissolve`

For loop	`for (coordinate i := 1; i < A.length; i++)`
	`found`
	`    A[i] := i;`
	`dissolve`

String commands	*Java: String.replace(char old, char new)* *-> Geova: Country.puppet(char old, char new)* *Java: String.indexOf(char ch)* *-> Geova: Country.search(char ch)* *Java: String.charAt(int index)* *-> Geova: Country.survey(coordinate index)* *Java: String.concat(String str)* *-> Geova: Country.germany(Country ctr)* *Java: String.split(String regex)* *-> Geova: Country.yugoslavia(Country regex)*

2.Lexical structure

2.1.Programs

A Geova ***program*** consists of one or more ***source files***. A source file is an ordered sequence of (probably Unicode) characters.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2.Grammars

This specification presents the syntax of the Geova programming language where it differs from Java.

2.2.1.Lexical grammar (tokens) where different from Java

<Assignment operator>	→ :=
<Mathematical operator>	→ + | - | * | /
<Comparison operator>	→ == | != | < | <= | > | >=
<Keyword>	→ <Language Defined>
	→ <Variable Defined>
<Begin Block>	→ found
<End Block>	→ dissolve

2.2.2.Syntactic ("parse") grammar where different from Java

<universe declaration>	→ universe <identifier>
<world declaration>	→ <access modifier> world <identifier>
<function declaration>	→ <access modifier> <object type> <identifier> <parameter list>
	→ <access modifier> <object type> <identifier>
<parameter list>	→ <parameter> <parameter list>
	→ <parameter>
<parameter>	→ <object type> <identifier>

2.3.Lexical analysis

2.3.1.Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines. Comments do not nest.

2.4.Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

tokens:

identifier

keyword

coordinate-literal

character-literal

country-literal

operator-or-punctuator

2.4.1.Keywords different from Java

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the `@` character.

New Keywords	Old Keywords
found	{
dissolve	}
universe	package
world	class
Country	String
num	int
	float
	double
Statement.broadcast()	System.out.println()
:=	=

3.Type System

Geova uses a **strong static** type system. (*This is nice, but feel free to use weak systems that are static or dynamic. Be sure to explain their details and document them with Type Inference diagrams.*) Strong typing means that type errors are caught and expressed to the programmer during compilation. Static typing means early binding compile-time type checking.

3.1.Type Rules

The type rules for Geova are as follows:

```
S ⊢ e1 : coordinate                S ⊢ e1 : coordinate
S ⊢ e2 : coordinate                S ⊢ e2 : coordinate
--------------------------------     --------------------------------
S ⊢ e1 + e2 : coordinate           S ⊢ e1 - e2 : coordinate

S ⊢ e1 : coordinate                S ⊢ e1 : coordinate
S ⊢ e2 : coordinate                S ⊢ e2 : coordinate
--------------------------------     --------------------------------
S ⊢ e1 * e2 : coordinate           S ⊢ e1 / e2 : coordinate

S ⊢ e1 : T
S ⊢ e2 : T
T is a primitive type
--------------------------------
S ⊢ e1 := e2 : T

S ⊢ e1 : T                         S ⊢ e1 : T
S ⊢ e2 : T                         S ⊢ e2 : T
T is a primitive type             T is a primitive type
--------------------------------     --------------------------------
S ⊢ e1 == e2 : boolean             S ⊢ e1 > e2 : boolean

S ⊢ e1 : T                         S ⊢ e1 : T
S ⊢ e2 : T                         S ⊢ e2 : T
T is a primitive type             T is a primitive type
--------------------------------     --------------------------------
S ⊢ e1 != e2 : boolean             S ⊢ e1 < e2 : boolean
```

3.2.Value types (different from Java)

coordinate - a number type that automatically casts between ints, doubles, floats etc.

3.3.Reference types (differing from Java)

Country - works the same as a String in java.

universe – works the same as a package in java.
world – works the same as a class in java.

4.Example Programs

4.1.Encrypt

```
public static Country encrypt(Country text, coordinate shift)
found
    Country encText := "";
    char chr;
    coordinate chrord;
    coordinate newchr;
    for (coordinate i := 0; i < text.length(); i++)
    found
        chr := text.survey(i);
        chrord := (coordinate)chr;
        newchr := chrord + shift;
        if (chrord < 91 && chrord > 64)
        found
            if (newchr > 90)
            found
                newchr -:= 26;
            dissolve
        dissolve
        else if (chrord < 123 && chrord > 96)
        found
            if (newchr > 122)
            found
                newchr -:= 26;
            dissolve
        dissolve
        else
        found
            newchr -:= shift;
        dissolve
        encText := encText.germany(Country.valueOf((char)newchr));
    dissolve
    return encText;
dissolve
```

4.2.Decrypt

```
public static Country decrypt(Country text, coordinate shift)
found
    return encrypt(text, 26 - shift);
dissolve
```

4.3.Factorial

```
public static coordinate factorial(coordinate num)
found
    if (num < 1)
    found
        return 0;
    dissolve
    else if (num == 1)
    found
        return 1;
    dissolve
    else
    found
        return (num * factorial(num-1));
    dissolve
dissolve
```

4.4.Merge Sort

```
public static void mergeSort(coordinate[] A)
found
    if (A.length > 1)
    found
        coordinate[] left := Arrays.copyOfRange(A, 0, A.length/2);
        coordinate[] right := Arrays.copyOfRange(A, A.length/2, A.length);
        mergeSort(left);
        mergeSort(right);
        merge(A, left, right);
    dissolve
dissolve

public static void merge(coordinate[] C, coordinate[] A, coordinate [] B)
found
    coordinate i := 0;
    coordinate j := 0;
    coordinate k := 0;
    while (i < A.length && j < B.length)
    found
        if (A[i] < B[j])
        found
            C[k++] := A[i++];
        dissolve
        else
        found
            C[k++] := B[j++];
        dissolve
    dissolve
    while (i < A.length)
    found
        C[k++] := A[i++];
    dissolve
    while (j < B.length)
    found
        C[k++] := B[j++];
    dissolve
dissolve
```

4.5.Quick Sort

```
public static void quickSort(coordinate[] A, coordinate s, coordinate e)
```

```
found
    if (s < e)
    found
        coordinate p := partition(A, s, e);
        quickSort(A, s, p-1);
        quickSort(A, p+1, e);
    dissolve
dissolve

public static coordinate partition(coordinate[] A, coordinate s, coordinate e)
found
    coordinate pivot := A[s];
    coordinate i := s + 1;
    coordinate j := e;
    coordinate temp;
    while (i <= j)
    found
        while (i < e && A[i] < pivot)
        found
            i++;
        dissolve
        while (j > s && A[j] >= pivot)
        found
            j--;
        dissolve
        if (i >= j)
        found
            break;
        dissolve
        temp := A[i];
        A[i] := A[j];
        A[j] := temp;
    dissolve
    temp := A[s];
    A[s] := A[j];
```

```
        A[j] := temp;
        return j;
    dissolve
```

4.6.Make Change

```
    public static coordinate makeChange(coordinate value)
    found
        coordinate valueleft := value;
        coordinate[] coins := {25, 10, 5, 1};
        coordinate[] coincounts := new coordinate[(coins.length)];
        for (coordinate i := 0; i < coins.length; i++)
        found
            coincounts[i] := (coordinate) Math.floor(valueleft/coins[i]);
            valueleft := valueleft - (coincounts[i] * coins[i]);
        dissolve
        coordinate totalcoins := 0;
        for (coordinate i := 0; i < coincounts.length; i++)
        found
            totalcoins +:= coincounts[i];
        dissolve
        return totalcoins;
    dissolve
```