Paradigmas de la programación:

Tipados en lenguajes:

Python:

Es de tipado dinámico pues una variable puede tomar valores de distinto tipo en distinto momento. Esto queda explicitado en el siguiente código:

```
# encoding: utf-8

def tipado_dinamico():

    entero = 10
    print entero + entero
    entero = "hola"
    print entero + entero

tipado_dinamico()
```

Donde la primera salida en pantalla corresponde al primer valor-tipo de la variable:
→20

Y la segunda salida corresponde al segundo valor el cual sería, por ser dinámico:
→ holahola

De la misma forma,
Ruby:

Es de tipado dinámico.
El código utilizado fue exactamente igual al de python variando únicamente en la sintaxis.

Java:

Es de tipado estático pues suponiendo que este fuera dinámico, el siguiente código podría ejecutarse:

```
class TipadoFuerte {

        public static void main(String[] args)
        {
                int estatico = 0;
                /* Intentamos cambiar el tipo de
                la variable "estatico"*/

                estatico = "Soy un string";
                /* Esto dará un error en tiempo de compilación:
                Se esperaba un int. */
        }
}
```

Lo cual da un error en tiempo de compilación representado por la siguiente salida:

```
tipado.java:10: error: incompatible types
                estatico = "Soy un string";
                           ^
  required: int
  found:    String
1 error
```

y nos permite concluir que sintaxis de java exige que al declarar la variable se declare su tipo también y las operaciones válidas sólo se contemplan para este tipo.

Scala:

De la misma forma que java, Scala es de tipado estático.
La demostración nace del suponer por el absurdo:

Suponiendo que es dinámico, este código podría compilarse:

```
object Tipado {

        def main(args: Array[String]) {

                val estatico = 0

                /* Esto dará un error en tiempo de compilación:
                Se esperaba un int. Ademas de que no permite reescribir el valor */
                estatico = "Nunca sere este valor"

        }

}
```

Pero su salida es:

```
tipado.scala:9: error: reassignment to val
                estatico = "Nunca sere este valor"
                 ^
one error found
```

y por lo tanto, la conclusión es la misma impuesta para el lenguaje Java.

Asignación en lenguajes:

Python:

Es de asignación múltiple pues una variable puede tomar el valor que se desee las veces que desee (y hay que recalcar esto).

El siguiente código lo demuestra:

```
# encoding: utf-8

def asignacion_multiple():

    # Un string tiene sus operaciones mapeadas a su tipo
    string = "10"
    print string + string # contactena

    # Pero si es entero, tendrá otras y su resultado sera distinto
    string = 10
    print string + string # suma

asignacion_multiple()
```

El cual da como salida:

```
→ 1010
→ 20
```

de la misma forma,
Ruby es de asignación múltiple, en este aspecto, el código utilizado fue el mismo que en python pero con la sintaxis de Ruby y por lo tanto en estos términos, su salida es igual.

Java:

Es de asignación múltiple siempre y cuando se respete el tipo de la variable a reasignar.

```
class MultipleAsignacion {

    public static void main(String[] args)
    {

        int multipleasignacion = 0;
        /* Intentamos cambiar el valor
        de multipleasignacion */
        System.out.format("Valor inicial: %d%n", multipleasignacion);

        multipleasignacion = 10;
        System.out.format("Valor reasignado: %d%n", multipleasignacion);
        /* Podemos observar que esto se satisface
        siempre y cuando respete su tipo */
```

```
        }
}
```

La salida en pantalla de este código al momento de ejecución es:

→ Valor inicial: 0
→ Valor reasignado: 10

y esto demuestra la multipleasignación .

Scala:

Al igual que Oz, Scala es de asignación única.
Pues, suponiendo que fuera de múltiple asignación, el siguiente código compilaría:

```
object AsignacionUnica {

        def main(args: Array[String]) {

                val asignacionUnica = 0

                /* Esto dará un error en tiempo de compilación:
                Scala es de asignación única */
                asignacionUnica = 1

        }

}
```

Lo cual es falso pues su salida en tiempo de compilación es:

```
asignacion.scala:9: error: reassignment to val
                asignacionUnica = 1
                        ^
one error found
```

Esto hace explícito el error de reasignación.

Javascript:

También es de asignación múltiple.

El siguiente es el script en sí ignorando el código html necesario para poder validar su comportamiento:

```
<script type="text/javascript">
        var multiple_asignacion = 10;
        alert("Mi primera asignacion es: " + multiple_asignacion);
        multiple_asignacion = "un string";
        alert("Mi segunda asignacion es: " + multiple_asignacion);
</script>
```

Se puede contemplar finalmente que su salida siendo ejecutada desde un navegador:

→ Mi primera asignación es: 10
→ Mi segunda asignación es: un string

Alcance del lenguaje:

Python:

Tiene alcance estático pues los valores almacenados no cambian de la declaración inicial.
De modo contrario, siempre tomaría el último valor asignado.

Analicemos el siguiente código:

```
# encoding: utf-8

def execute():

    y = 2

    def Q(x):
        print "Estático pues: x+y=", x+y

    def rewrite():
        y = 3
        Q(2)
    rewrite()

execute()
```

Esto da como salida:

→ Estático pues: x+y= 4

En particular, **Ruby** es un poco más estricto pues no permite dejar una variable sin definir en cierto contexto.
Pero si todo está definido dentro de la clausula, se comprueba que Ruby **es estático**.

Veamos el siguiente código:

```
# encoding: utf-8

def execute()

   y = 222

   def Q(y)
      puts "Estático pues es y seguirá siendo"
   end

   def rewrite()
      y = 333
      Q(y)
   end

   rewrite()
   puts y

end
execute()
```

Lo cual da como salida:
→ Estático pues es y seguirá siendo
→ 222

con **Java** sucede **exáctamente lo mismo**. El valor de la variable no se sobreescribe por uno más actual pues de otra forma el valor que tomaría la variable siempre sería el mismo.

```
class Alcance {

	void q(int y) {
		System.out.format("Estático pues el valor de y es: %d\n", y);
	}

	void rewrite(int y){

		y = 3;
		q(y);
	}

	public static void main (String args[]) {

		int y = 2;
		Alcance a = new Alcance();
		a.rewrite(y);
		System.out.format("Realmente este: %d\n", y);
  }
}
```

Esto da como salida:
→ Estático pues el valor de y es: 3
→ Realmente este: 2

Scala es igual.

```
object Alcance {

	def q(y: Int) {
		println("Estático pues el valor de y es:\n" + y);
	}

	def rewrite(y: Int){
		val y = 3;
		q(y);
	}

	def main(args: Array[String]) {

		// val a = new Alcance()
		val y = 2
		rewrite(y)
		println("Realmente este: " + y)
	}
}
```

lo cual da como salida:
→ Estático pues el valor de y es: 3
→ Realmente este: 2

Finalmente y como excepción, **Javascript es de alcance dinámico**:

Pues basta analizar la salida del siguiente código:

```
	function execute() {
		y = 2;

		function Q(x) {

			var result;
			result = x + y;
			alert("Dinamico pues: x+y= " + result);

		}

		function rewrite() {

			y = 3;
			Q(90)
		}
		rewrite();
	}
	execute();
```

Donde su salida es:

→ Dinamico pues: x+y= 93
Claramente 'y' toma el valor más cercano a la llamada rewrite.

Tipado de lenguajes(fuerte y débil):

C es de tipado fuerte
pues no permite operaciones entre tipos distintos, además del pre requisito de declarar el tipo de la variable.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {

	int fuerte = 0;
	char *string = "Esto es un string";
	char *resultado = "Nunca tomare este valor";
	/* La definición de estático determina que
	si el tipo ya está definido, SÓLO podemos
	realizar operaciones definida para este tipo
	a menos que la conversión a -otro- tipo sea
	explícita: */

	resultado = string * fuerte;

	printf("%s\n", resultado);

	return EXIT_SUCCESS;

}
```

Este código genera un error en tiempo de compilación:
clang tipado.c

```
tipado.c:15:21: error: invalid operands to binary expression ('char *' and 'int')
	resultado = string * fuerte;
		    ~~~~~~ ^ ~~~~~~~
1 error generated.
```

Contrastando con **Javascript** el cual **es debilmente tipado:**

```
function tipado () {
	var debil = "¡Quiero duplicarme!\n";
	// ¡Concedido!
	debil = debil * 2;
	alert(debil);
	debil = "¡Quiero abrazar un número!";
	// ??? Ok..
	debil += 222;
	alert(debil);
}
tipado();
```

Donde se puede observar que si bien no todas las operaciones están definidas (multiplicar strings por lo visto no), las operaciones entre distintos tipos está permitida.

La salida de esto es:
→ NaN
→ ¡Quiero abrazar un número!222

Recursión:

A continuación se realizan dos funciones recursivas a la cola mutuamente no optimizadas por el compilador escritas en Scala:

```
object Recursivo {
	def main(args: Array[String]) {
		// Devuelve true si es par
		def EsPar (x:Int) : Boolean =
			if (x==0) true
			else EsImpar(x-1)

		// Devuelve true si es impar
		def EsImpar (x:Int) : Boolean =
			if (x==0) false
			else EsPar(x-1)

		// Llamamos a las funciones
		def arg1 = 1
			def result1 = EsImpar(arg1)
				println("EsImpar(" + arg1 + ")= " + result1) //aca se fija si es impar el arg

		def arg2 = 1
			def result2 = EsPar(arg2)
				println("EsPar(" + arg2 + ")= " + result2 + "\n") //aca se fija si es par el arg

		def arg3 = 100
			def result3 = EsImpar(arg3)
				println("EsImpar(" + arg3 + ")= " + result3)

		def arg4 = 100
			def result4 = EsPar(arg4)
				println("EsPar(" + arg4 + ")= " + result4)
	}
}
```

Al momento de compilar se puede observar el considerable tiempo que demora. De la misma forma, su ejecución y salida:

→ EsImpar(1)= true
→ EsPar(1)= false
→
→ EsImpar(100)= false
→ EsPar(100)= true