Babel

Eduardo Amaya Espinosa y Pablo Sanz Sanz

Marzo de 2020

1. Introducción

Babel es el lenguaje de programación que hemos diseñado y que aquí detallamos. La elección del nombre viene porque hemos seleccionado las características que más nos gustaban de distintos lenguajes y las hemos fusionado para crear el nuestro. Nuestro objetivo es tener un código intuitivo, fácilmente legible y estructurado, sin olvidar nunca la funcionalidad y que el tiempo que tenemos para implementarlo es limitado, por lo que hemos decidido prescindir de algunos detalles superfluos.

En este documento explicamos los detalles principales del lenguaje con descripciones verbales y con ejemplos de código que ayudarán al lector a comprender y familiarizarse con el lenguaje. En general nos centramos en tres grandes bloques: identificadores y ámbitos, tipos e instrucciones. En estos tres bloques se detallan todas las elecciones que hemos hecho y los beneficios que nos proporcionan. Por último concluimos con una breve descripción de la gestión de errores que tendrá nuestro compilador, aspecto clave en cualquier lenguaje de programación que se precie.

2. Identificadores y ámbitos

2.1. Variables

Las variables deberán ser declaradas explícitamente, pudiendo asignarles un valor en el momento de la declaración. Si queremos declarar una constante usaremos la palabra reservada `const` antes de indicar el tipo de la constante. Las constantes deben declararse con un valor asignado, que no podrá ser modificado más adelante.

Los nombres de las variables admiten cualquier carácter alfanumérico, además de barras bajas (`_`), pero deben comenzar siempre por una letra minúscula o barra baja. Sólo hay un caso especial, las constantes globales, que sólo pueden contener letras mayúsculas o barras bajas. Sin embargo, `_` no es válido como identificador, pues es una palabra reservada, como veremos más adelante. En general, ninguna palabra reservada, como `const`, o cualquiera de las que vayan apareciendo en el futuro puede ser utilizada como identificador.

Además, el lenguaje diferencia entre mayúsculas y minúsculas, por lo que sería válido llamar a una variable `cOnst`, por ejemplo, pero `cOnst` y `cOnSt` no harían referencia a la misma variable.

El operador de asignación es el operador `=`.

```
1 Int x;                    // a partir de ahora existe la variable x
2 x = 0;                    // x tiene valor 0
3 y = 1;                    // error: y no existe
4 Int y = 5;                // ahora si, creamos 'y' y le asignamos el valor 5
5 Int _soyUnaVariable_1 = 1; // valido
6 Int X;                    // no valido: empieza por mayuscula
7 Int _;                    // no valido: palabra reservada
```

2.2. Bloques

Disponemos de una forma de crear bloques de forma que las variables que se crean en un bloque no son accesibles desde fuera del mismo. Además, los bloques permiten ocultar variables exteriores al bloque con el mismo nombre, pero sin perder su valor. Para crear bloques usamos las llaves: `{/* bloque */}`.

```
1  Int x = 0;
2  {
3      // x == 0
4      x = 5;
5      // x == 5
6      Int x = 10;
7      // x == 10
8      Int y = 0;
9      // x == 10, y == 0
10 }
11 // x == 5, y no existe
```

2.3. Funciones y procedimientos

En nuestro lenguaje también vamos a disponer de funciones y procedimientos. No vamos a diferenciar entre ellos y se van a tratar todas como funciones. Los procedimientos van a devolver un tipo especial `Void`, que no va a ser utilizado más que para unificar funciones y procedimientos. Sería válido, sin embargo, asignar el valor devuelto por un procedimiento a una variable de tipo `Void`, pero después no podremos hacer gran cosa con ella.

Para declarar una función utilizaremos la sintaxis `Tipo nombre(parámetros) {/* código */}`, donde los parámetros, en caso de tenerlos, aparecen separados por comas indicando primero su tipo y luego el identificador asociado. Como vemos, cada función tiene su propio bloque asociado con todo lo que esto implica en cuanto a visibilidad.

Los identificadores de función siguen las mismas reglas que los de variable. Las funciones se diferencian por su nombre y tipo de sus parámetros por orden, por lo que se podrán declarar varias funciones con el mismo nombre siempre y cuando difieran en sus parámetros. Por ejemplo, `Int foo(Int x)`, `Int foo()` y `Void foo(Real x)` son funciones distintas, pero `Void foo()` nos produciría un conflicto de nombres con la segunda declaración.

En cualquier punto de la función se puede devolver un valor al punto de llamada de la misma con una cláusula `return` seguida de lo que se quiera devolver. Toda función debe asegurar que todas sus ramas tienen un punto de salida. El caso de los procedimientos es un poco especial y admite que no se indique ningún return o que en caso de tenerlo no se devuelva ningún valor (`return;`). En tales casos se considera implícitamente que se tiene `return nothing;`.

Las funciones sólo podrán ser declaradas en el *scope* global, no permitiéndose el anidamiento de funciones. Hay una función especial `Void main()` que será el punto de entrada a todos nuestros programas y que debemos declarar siempre para que un programa comience a funcionar.

```
// funcion sin argumentos
Int foo() {
    return 1;
}

// funcion con dos argumentos
Int bar(Int a, Int b) {
    return a * b;
}

// procedimiento con un argumento
Void print(Int x) {
    /*
        codigo para mostrar por pantalla
        no hace falta return
    */
}

Void main() {
    // punto de entrada: el programa comienza aqui
    Int x = foo();      // llamada a funcion sin argumentos
    Int y = bar(x, 2);  // llamada a funcion con argumentos
    print(x + y);       // llamada a procedimiento con un argumento
}
```

2.4. Módulos e importación

Terminamos esta sección hablando de los módulos.

Cuando tenemos programas más complicados es útil dividir las funcionalidades en diferentes ficheros, de forma que el código queda mejor organizado. Estas divisiones son los diferentes módulos de nuestro programa. Para poder utilizar desde un fichero funciones, variables, etc., que tenemos definidas en otro fichero podemos utilizar las cláusulas `import`. Estas deben aparecer en las primeras líneas de un fichero y su función será la de incluir el fichero indicado en el actual. El fichero se indicará a partir de su ruta relativa en el sistema de ficheros.

```
// ./util/math.bbl
const Real pi = 3.1416;

Int add(Int x, Int y) {
    return x + y;
}

Int mult(Int x, Int y) {
    return x * y;
}

// ./main.bbl
import util/math; // se asume la extension del archivo

Void main() {
    // podemos acceder a ellas como si estuvieran en main.pl
    print(mult(add(1, 2), 5));
    print(pi);
}
```

3. Tipos

3.1. Tipos básicos

Nuestro lenguaje dispone de una serie de tipos básicos o primitivos a disposición del usuario, que nos serán útiles luego para formar otros tipos más complejos. Estos tipos básicos son los siguientes:

- `Int` (4 B). Números enteros comprendidos entre $+2\,147\,483\,647$ y $-2\,147\,483\,648$. Se pueden indicar en su codificación decimal, binaria (`0bnn...nn`), octal (`0nn...nn`) o hexadecimal (`0xn...nn`). Por tanto, no se deben representar enteros que comiencen por 0, pues se entiende que están representados en octal.
- `Real` (4 B). Números reales comprendidos entre $3{,}402\,823\,5 \cdot 10^{38}$ y $1{,}4 \cdot 10^{-45}$ en valor absoluto. Se representan en notación decimal, separando los decimales por un punto.
- `Bool` (1 B). Tipo verdadero o falso, con dos posibles valores: `true` y `false` .
- `Char` (1 B). Tipo formado por los 256 caracteres. Un carácter se expresa entre comillas simples, por ejemplo, `'a'` .
- `Void` (1 B). Tipo "vacío" cuyo único valor es `nothing` , pero no tiene operaciones. Útil para unificar funciones y procedimientos.

Los tipos básicos tienen la característica de que se representan en memoria estática tal cual son y se copian sus valores en llamadas a función. Es decir, tendremos una representación en memoria del valor de una variable de tipo primitivo y al pasarla como parámetro a una función este valor se copiará, de forma que el retornar de la función aseguramos que se mantiene el valor previo a la llamada.

```
1  Int next(Int n) {
2      n = n + 1;
3      return n;
4  }
5
6  Void main() {
7      Int x = 0;
8      print(next(x)); // 1
9      print(x);       // 0: conserva su anterior valor
10 }
```

3.2. Arrays

Cuando tenemos varias variables relacionadas entre sí disponemos del tipo cualificado `Array` . Este nos permite almacenar numerosos elementos de un mismo tipo uno tras otro. Para cualificar un array con un tipo concreto usamos el operador diamante: `Array<Int>` , por ejemplo.

Para crear un array podemos usar o bien el generador `[,]` , o bien el constructor predeterminado, al que únicamente se le indica la capacidad y reserva el espacio en memoria. En este segundo caso no se asegura nada acerca del contenido del array y en caso de ser de un tipo no básico estos valores se considerarán `null` . La sintaxis es `Array(n1, ..., nk)` , donde cada `ni` es el tamaño del array en la dimensión i (cada array "dentro" de otro array representa una dimensión). Si el array tiene n dimensiones y $k < n$ se inicializarán los tamaños de las primeras k dimensiones la dimensión k estará formada por `null` s. Aunque un array multidimensional se inicialice con un tamaño concreto en una dimensión, asignando un nuevo array de distinta longitud a esa dimensión podemos modificarla sin problema.

Como atajo para los casos en los que se quiera crear un array tan sólo especificando un tamaño para asignarlo a una variable podemos usar esta misma sintaxis de constructor antes del identificador de la variable.

Es decir, tras indicar que la variable que se va a crear es de tipo `Array<?>` se especifica entre paréntesis el tamaño de las dimensiones y después se le da nombre a la variable.

Para acceder a los elementos del array usaremos el operador corchete (`[n]` , donde `n` es un entero mayor o igual que cero que indica el índice al que se accede). El primer elemento se encuentra en la posición 0, el segundo en la 1, etc.

Los arrays tienen una longitud fija e intentar acceder a un elemento fuera de su rango (ya sea negativo o mayor o igual que la longitud) dará un error en tiempo de ejecución. Utilizando el atributo `size` de un array podemos conocer su longitud.

```
Array<Int> arr = [1, 2, 3, 4, 5]; // creacion en el momento
Array<Array<Int>> matrix = Array(3); // array bidimensional con 3 filas
matrix[0] = arr;
print(matrix.size); // 3
print(matrix[0].size); // 5
print(matrix[1]); // null
print(matrix[3]); // error
Array<Int>(7) row; // atajo para array de 7 elementos
matrix[1] = row; // cada fila puede tener una longitud distinta
```

A diferencia de los tipos básicos, un array únicamente es una referencia al primer elemento de dicha colección. Es decir, ocupa exactamente lo que una dirección de memoria. Los elementos del array se almacenan uno tras otro en la memoria dinámica de nuestro programa. Por tanto, una llamada a función con un array como parámetro puede modificar permanentemente los elementos del array. Sin embargo, no va a ser capaz de cambiar la referencia de nuestro array, porque esta sí que es copiada (al fin y al cabo es como un entero).

```
Void set(Array<Int> a, Int i, Int x) {
    a[i] = x;
}

Void newArray(Array<Int> a, Int n) {
    a = Array(n);
}

Void main() {
    Array<Int> array = [1, 2, 3, 4, 5];
    print(array[2]); // 3
    set(array, 2, 0);
    print(array[2]); // 0: el elemento se ha modificado
    newArray(array);
    print(array); // [1, 2, 0, 4, 5]: la referencia no
}
```

Por último, un tipo especial de array es `Array<Char>` , que habitualmente suele usarse para representar cadenas de texto. Por ello existe una alternativa para este tipo: `String` . Dispone de una declaración especial, delimitando la cadena con dobles comillas. Por ejemplo, `"Hola" == ['H', 'o', 'l', 'a']` .

3.3. Objetos anónimos

Cuando queremos almacenar diferentes valores en una misma estructura – pero un array no nos sirve porque, o bien son de distinto tipo, o bien es necesaria algún tipo de organización más allá de darle un valor numérico a cada elemento –, disponemos de los objetos anónimos.

Un objeto anónimo es una estructura algo más compleja que almacena valores de diferentes tipos asignando un nombre a cada valor, sin ningún orden concreto. Estos valores se denominan atributos. En cuanto a

almacenamiento en memoria siguen unas reglas parecidas a las de los arrays, con lo cual se usarán también punteros al objeto en memoria dinámica.

Dos objetos anónimos serán iguales si son idénticos, es decir, la misma dirección de memoria, o si sus atributos son iguales, entendido como igualdad de nombre de los atributos e igualdad de valor.

Para declarar objetos anónimos usaremos las llaves y dentro de ellas especificaremos sus cero o más atributos separados por comas y para cada uno de ellos su tipo y su valor. Más concretamente:

`{ Tipo1 atributo1 = valor1, Tipo2 atributo2 = valor2, ... }`.

Todos los atributos deben tener algún valor inicial, aunque luego podrá ser modificado. Pero una vez creado un objeto anónimo ya no podremos añadirle más atributos.

El tipo utilizado para un objeto anónimo será `Form`. Para acceder al valor de un atributo y leerlo o modificarlo usaremos el operador `.` seguido del nombre del atributo al que se quiere acceder.

```
Form student = {
    String name = "John",
    Int age = 22,
    Real height = 1.83
};
print(student.name); // John
print(student.surname); // error: surname no existe
student.age = 50;
print(student.age); // 50
student.surname = "Doe"; // error: surname no existe
Form teacher = {
    String name = "John",
    Int age = 50,
    Real height = 1.83
};
print(teacher == student); // true
Form origin = { Int x = 0, Int y = 0 };
print(student == point); // false
// Podemos tener objetos complejos
Form tree = {
    Form left = { Form left = {}, Int value = 0, Form right = {} },
    Int value = 3,
    Form left = { Form left = {}, Int value = 5, Form right = {} }
};
```

Como los objetos anónimos pueden tener cualquier forma concreta y en muchas ocasiones no se podría determinar si un atributo accedido tiene el tipo esperado o siquiera existe, van a tener una serie de restricciones importantes en su uso.

No está permitido su uso como tipo de una función, es decir, las funciones no pueden devolver el tipo `Form`. En general, se aplica esta restricción a cualquier otro tipo de instrucción que devuelva algún valor. De la misma forma, tampoco podrán ser utilizados como parámetros de ninguna función ni pueden ser declarados sin recibir ningún valor inicial.

En definitiva, cualquier intento de uso de los objetos anónimos que implique no conocer sus atributos o sus tipos no estará permitido.

3.4. Clases

Los tipos anónimos son útiles para agrupar datos cuando se está trabajando con una gran cantidad de variables o para tener mayor organización o jerarquía en las variables globales. Sin embargo, tienen grandes limitaciones, como ya hemos estado viendo.

Así, para añadir un mayor nivel de abstracción al lenguaje y permitir más flexibilidad aparecen las clases. Al igual que los objetos anónimos, las clases agrupan atributos comunes a un mismo tipo de objeto. Sin embargo, hay tres notables diferencias con respecto de los objetos anónimos.

La primera es que las clases sí que tienen nombre. Así, todos los objetos de un mismo tipo pertenecerán a una misma clase y podremos asegurar cuáles son sus atributos y sus tipos en tiempo de compilación sin ningún problema. Esto nos permite usarlos en todo tipo de instrucciones con las mismas restricciones que pueda tener un tipo básico.

La segunda diferencia con los objetos anónimos es que las clases admiten métodos además de atributos. Un método es una función con un parámetro implícito `this` que hace referencia al propio objeto. Por tanto, igual que con los objetos anónimos, podemos acceder a cualquier atributo o método del propio objeto utilizando `this.nombre`. `this` es una palabra reservada y no hace falta incluirlo entre los parámetros del método.

La última diferencia aparece con la visibilidad de los atributos y los métodos. En los objetos sin nombre todos los atributos eran públicos, es decir, podían utilizarse desde el exterior del objeto. En cambio, los atributos de las clases van a ser todos privados, por lo que ninguna instrucción ajena a la clase va a poder acceder a ellos: ni leerlos ni modificarlos. Los métodos, sin embargo, serán todos públicos.

De esta forma una clase proporcionará una serie de funcionalidades que se puedan usar desde el exterior, pero no permitiendo una alteración no deseada del estado del objeto. Una forma de hacer visibles ciertos atributos a elección es utilizando métodos *getters* o *setters*.

Las clases se declaran en el entorno global, como las funciones, y no está permitido declararlas en ningún otro lugar. Tienen un nombre que sigue las mismas reglas que los identificadores de variable, con la excepción de que los nombres de clase deben comenzar por una letra mayúscula. La sintaxis de declaración es

`class Nombre { [Atributos] [Constructores] [Métodos] }`.

Los atributos se declaran de la misma forma que las variables y pueden ser de cualquier tipo o clase (en el caso de `Form` respetando sus restricciones). Los constructores se declaran de la misma forma que las funciones, pero no especifican tipo y su nombre es siempre la palabra clave `constructor`. El constructor debe dar valor a los atributos del objeto y para acceder a ellos utilizará `this`. No debe devolver nada. Por último, los métodos se declaran exactamente igual que las funciones.

Una vez definida una clase podemos crear objetos de la misma con diferentes valores para sus atributos. Para darles un valor inicial usaremos un constructor de la clase, que recibirá unos parámetros, "construirá" el objeto y lo devolverá como resultado. La llamada al constructor se hará como cualquier llamada a función, pero utilizando el nombre de la clase como "nombre" de la función.

De la misma forma que con los arrays existe un atajo para los casos en que queramos crear un objeto y asignárselo a una variable. Se hará con el operador paréntesis tras el nombre del tipo y dentro de estos aparecerán los argumentos del constructor.

Y también de forma parecida a los arrays aparece el puntero `null`, en este caso cuando declaramos un objeto sin inicializarlo con un constructor.

Finalmente, a diferencia de los objetos anónimos, dos objetos de una misma clase serán iguales si son exactamente el mismo objeto o, en caso de disponer la clase de un método `Bool _equals`, si este devuelve `true`. Por tanto, ya no se comprueba atributo a atributo, sino que dependerá de cada clase concreta. Asimismo, tampoco será posible comparar un objeto de una clase con uno de otra ni con un objeto anónimo aunque tengan la misma estructura interna.

point.bbl

```
// En un fichero separado para mejor formato del texto, pero no necesario
class Point {
    // Atributos
    Int x;
    Int y;

    // Constructores
    constructor() {
        // Por defecto el origen
        this.x = 0;
        this.y = 0;
    }
    constructor(Int x, Int y) {
        // Se pueden usar los mismos nombres para los parametros porque los atributos se
        // obtienen con this
        this.x = x;
        this.y = y;
    }

    // Metodos
    Real norm() {
        return sqrt(this.x * this.x + this.y * this.y);
    }
    Bool _equals(Point p) {
        // Podemos acceder a los atributos de p porque es de la misma clase en la que estamos
        return this.y == p.y;
    }
}
```

main.bbl

```
import point;

Void main() {
    Point origin = Point(); // constructor sin parametros: (0, 0)
    Point(1, 0) ox; // atajo, constructor con parametros: (1, 0)
    print(ox.norm()); // llamada a metodo: 1.0
    Form p = { Int x = 1, Int y = 0 };
    print(ox == p); // error: no son comparables
    print(ox == origin); // true: equals no usa x
    print(origin.x); // error: los atributos no sin visibles desde fuera de la clase
    Point oy;
    print(oy); // null: oy no se ha construido aun
}
```

4. Conjunto de instrucciones

4.1. Instrucciones básicas

4.1.1. Asignación

A las variables se les puede asignar un valor en el momento de la declaración o en cualquier otro momento. Para asignarles un valor simplemente hay que indicar el nombre de la variable, seguido del operador de asignación `=` y el valor que se le quiera asignar. El valor que se le asigne puede ser cualquier expresión siempre que esta tenga el mismo tipo que la variable.

```
Int x = 0;
Int y = 15 - (12 + 3);
// x == y
Char a = 'a';
Real z;
z = y - 3.2;
Array<Int> list = Array(5);
list = [1,2,3,4];
// list == [1,2,3,4]
Bool t = true && false;
```

4.1.2. Llamadas a procedimientos, funciones y métodos

Para llamar a las funciones simplemente debemos escribir su nombre y, entre paréntesis, cada uno de los argumentos que queremos enviarle en el orden en que están indicados. Para recuperar el valor devuelto por una función no tenemos más que asignarlo a una variable de su mismo tipo o utilizarlo en una expresión.

```
Int foo() {
    return 1;
}

Int sum(Int a , Int b) {
    return a + b;
}

Char toChar(Bool b) {
    if b {
        return 't';
    }
    return 'f';
}

Void print(Int x) {
    /*
        codigo para mostrar por pantalla
        no hace falta return
    */
}

Void main() {
    Int x = sum(foo(), 2); // x == 3
    Bool b = toChar(1 == 1); // error de tipos
    Char c = toChar(1 == 1); // c == 't'
    sum(x, 4); // no da error, pero el valor devuelto se pierde
    print(sum(x, 4)); // se imprime por pantalla el numero 7
    Void p = print(4); // es correcto, pero p no sirve para nada
}
```

4.1.3. Condicionales

La instrucción condicional principal de nuestro programa es el conjunto `if` - `else`. Una instrucción `if` vendrá siempre seguida de un valor de tipo `Bool` (que puede ser cualquier expresión de este tipo) y de un bloque de código. El bloque se ejecutará si y solo si el valor booleano es equivalente a `true`. Una instrucción `if` puede ir seguida de otra instrucción `else` que a su vez irá seguida de otro bloque de código. Este bloque se ejecutará si y solo si el booleano es equivalente a `false`.

Además, existe la posibilidad de anidar varias instrucciones condicionales. Esto se hace incluyendo un `if` inmediatamente después de un `else`. El siguiente bloque de código se ejecutará si y solo si el booleano que sigue al `if` es `true`. Si existen varias instrucciones condicionales anidadas solo se ejecutará el bloque

correspondiente al primer `if` cuyo booleano sea `true`. En caso de llegar a una instrucción `else` que no vaya seguida de ningún `if` y no haber sido ninguno de los booleanos anteriores `true`, se ejecutará el bloque correspondiente a ese `else`.

```
Int greater(Int a, Int b) {
    if a > b {
        return a;
    } else {
        return b;
    }
}

Int lower(Int a, Int b) {
    if a <= b {
        return a;
    } else {
        return b;
    }
}

Int compare(Int a, Int b) {
    if a > b {
        return 1;
    } else if a < b {
        return -1;
    } else {
        return 0;
    }
}

Void main() {
    Int a = 5;
    Int b = 2;
    Int x = greater(a,b); // x == 5
    x = lower(a,b); // x == 2
    x = compare(a,b); // x == 1
    x = compare(b,a); // x == -1
    x = compare(a,a); // x == 0
}
```

Por último, tenemos la instrucción `switch`. Esta instrucción viene seguida de una variable de un tipo primitivo. Tras la variable vienen varios valores del mismo tipo y cada valor viene seguido de un bloque de código. Solo se ejecutará un bloque de código si el valor que lo precede es igual que el valor de la variable. Por último se puede añadir un valor `_` cuyo bloque se ejecutará si ningún otro valor coincide con el de la variable.

```
Int abcdToInt(Char a) {
    switch a {
        'a' -> {
            return 1;
        }
        'b' -> {
            return 2;
        }
        'c' -> {
            return 3;
        }
        'd' -> {
            return 4;
        }
        _ -> {
            return 0;
        }
    }
}

Void main() {
    Int x = abcdToInt('a'); // x == 1
    x = x + abcdToInt('c'); // x == 4
    x = abcdToInt('Z'); // x == 0
}
```

4.1.4. Bucles

En nuestro lenguaje hay dos tipos de bucle, el bucle `while`, que nos servirá para implementar cualquier programa iterativo, y el bucle `for`, que utilizaremos sólo para recorrer un objeto iterable. El `while` irá seguido de una expresión booleana que se evaluará en cada iteración y un bloque de código que se ejecutará siempre que la condición sea `true`. La primera vez que al evaluar la condición se obtenga `false` se terminará con la ejecución del bucle y se pasará a la siguiente instrucción.

En cuanto al bucle `for`, este irá seguido de un nombre de variable (que no tiene que estar declarada previamente), la palabra clave `in` y un objeto iterable. Así, la variable irá recorriendo el objeto mientras no llegue al final del mismo y en cuanto llegue al final se terminará el bucle y se pasará a la siguiente instrucción. Además añadimos el operador `...` que es un operador infijo binario que dados dos enteros a y b devuelve el intervalo cerrado de enteros $[a, b]$ como objeto iterable.

```
Int i = 0;
while i < 10 {
    i = i + 1;
}
// i == 10

Array<Int> list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
Int j = 1;
for Int x in list {
    j = j * x;
}
// j == 10!

Int k = 1;
for Int y in 1...10 {
    k = k * y;
}
// k == 10!
```

4.2. Operadores

Entre los operadores binarios de este lenguaje están, por supuesto, los operadores aritméticos básicos `+`, `-`, `*` y `/`; para la suma, resta, multiplicación y división respectivamente, además del `-` unario para el cambio de signo. Como complemento a la división se añade el operador `%`, que hace el resto de la división entera. A continuación, encontramos los operadores de comparación `>` (mayor), `<` (menor), `>=` (mayor o igual), `<=` (menor o igual), `==` (igual), `!=` (no igual) y `===` (que devuelve `true` si y solo si los dos operandos son el mismo objeto). Después tenemos los operadores lógicos `&&` y `||` que representan la conjunción y la disyunción lógica, además del operador unario prefijo `!` que es la negación lógica.

```
Int x = 10;
Int y = 2;
Int z = x + y; // z == 12
z = x - y; // z == 8
z = -z // z == -8
z = x * y; // z == 20
z = x / y; // z == 5
z = x % y; // z == 0

Bool b = x > y; // b == true
b = x < y; // b == false
b = x >= (x - 0); // b == true
b = (x + 1) <= x; // b == false
b = x == 10; // b == true
b = x != 10; // b == false
b = x === 10; // b == flase
b = x === x; // b == true

b = true && false; // b == false
b = true && (true || false); // b == true
b = !b // b == false;
```

Para terminar tenemos los operadores de asignación, además del `=`, visto más arriba, tenemos `+=`, `-=`, `*=`, `/=` y `%=`. Cada uno es equivalente a hacer la operación correspondiente entre dos operandos (siendo el primero de ellos una variable) y asignarle el resultado a dicha variable.

```
Int x = 4;
x += 3; // x == 7
x -= 3; // x == 4
x *= 3; // x == 12
x /= 3; // x == 4
x %= 3; // x == 1
```

Además de estos operadores principales, tenemos otros que llamamos "operadores especiales". Entre ellos destacan los corchetes `[]`, que se utilizan como operador para el tipo `Array`, tanto para crearlos como para acceder a un elemento. Al crearlos también se utiliza el operador `,` para separar los elementos. Para acceder a atributos dentro de los diferentes objetos disponemos del operador `.`, que se utilizará poniendo antes de él el nombre del objeto y después el nombre del atributo. Por último tenemos los paréntesis `()`, que se utilizan como estamos acostumbrados para modificar la prioridad de los operadores además de para pasar los parámetros a las funciones.

```
Array<Int> arr = [1, 2, 3, 4, 5]; // cada numero es un elemento, separados por comas
// y con corchetes al principio y al final

Form student = {
    String name = "Stephen",
    Int age = 32,
    Real height = 1.83
}; // elementos separados por comas

Int x = student; // error de tipos
Int x = student.age; // x == 32

Int y = 3 * 2 + 6 // y == 12
y = 3 * (2 + 6) // y == 24

class Point {
    Int x;
    Int y;

    constructor() {
        this.x = 0; // se accede al atributo x de si mismo.
        this.y = 0;
    }
}
```

El Cuadro 1 muestra los operadores ordenados por prioridad descendente y cada uno con su tipo, asociatividad y método que hay que implementar para sobrecargarlo (entre paréntesis el número de parámetros de dicho método).

Operador	Descripción	Tipo	Asociatividad	Método
()	Paréntesis (llamada de funciones)	Unario postfijo	Izquierda-a-derecha	
[]	Corchete (acceso a arrays)	Unario postfijo	Izquierda-a-derecha	
.	Punto (acceso a atributos y métodos)	Binario infijo	Izquierda-a-derecha	
+, −	Más, menos unarios (cambio de signo)	Unario prefijo	Derecha-a-izquierda	
!	Negación lógica	Unario prefijo	Derecha-a-izquierda	`_not(0)`
∗, /, %	Multiplicación, división, módulo	Binario infijo	Izquierda-a-derecha	`_mult(1)`, `_div(1)`, `_mod(1)`
+, −	Suma, resta	Binario infijo	Izquierda-a-derecha	`_plus(1)`, `_minus(1)`
<, <=	Menor, menor o igual relacionales	Binario infijo	Izquierda-a-derecha	`Bool _lt(1)`, `Bool _le(1)`
>, >=	Mayor, mayor o igual relacionales	Binario infijo	Izquierda-a-derecha	`Bool _gt(1)`, `Bool _ge(1)`
==, ! =, ===	Igual, no igual, identidad relacionales	Binario infijo	Izquierda-a-derecha	`Bool _equals(1)`
&&	Conjunción lógica	Binario infijo	Izquierda-a-derecha	`_and(1)`
||	Disyunción lógica	Binario infijo	Izquierda-a-derecha	`_or(1)`
...	Puntos suspensivos (intervalos)	Binario infijo	Izquierda-a-derecha	
=	Asignación	Binario infijo	Derecha-a-izquierda	
+ =, − =, / =, % =	Asignación de suma, resta, etc.	Binario infijo	Derecha-a-izquierda	*Con las definiciones de los respectivos*
,	Separador de expresiones	Binario infijo	Izquierda-a-derecha	

Cuadro 1: Prioridad de los operadores

4.2.1. Sobrecarga de operadores

Los operadores anteriores, que en principio están planteados para tipos básicos, pueden sobrecargarse para clases. Para ello el usuario deberá implementar dentro de la clase un método que reciba como parámetro (si es necesario) un objeto de la clase que corresponda y devuelva el resultado. Por ejemplo para sobrecargar la suma habrá que implementar un método `_plus`. Algunos casos son especiales, como en el caso de los operadores relacionales, que deben devolver siempre `Bool` y recibir como parámetro un objeto de la misma clase.

```
class Point {
    Int x;
    Int y;

    constructor()  {
        this.x = 0;
        this.y = 0;
    }

    constructor(Int x, Int y) {
        this.x = x;
        this.y = y;
    }

    Point _plus(Point other) {
        return Point(this.x + other.x, this.y + other.y);
    }

    Point _mult(Int number) {
        return Point(rhis.x * number, this.y * number);
    }
}

Void main() {
    Point p1 = Point(2, 5);
    Point p2 = Point(1, -2);
    Point result = p1 + p2;
    // result.x == 3
    // result.y == 3
    result *= 2;
    // result.x == 6
    // result.y == 6
}
```

5. Gestión de errores

Por último detallamos brevemente cómo va a gestionar nuestro compilador los errores que vaya encontrando.

Nuestro compilador comenzará su tarea en con el análisis léxico de todo nuestro código. Una vez hecho esto, se irá analizando el código y creando un árbol que representa el programa entero para, finalmente, traducirlo a la máquina-P. Durante este proceso podemos encontrarnos con varios tipos de errores.

Primero, es posible que hayamos cometido errores léxicos, es decir, hemos escrito algo que ni siquiera es válido. En tales casos el compilador nos informará de las líneas y la columna dentro de cada una donde haya encontrado tales errores, aunque algunos de ellos pueden ser consecuencia de uno anterior.

Una vez superada la fase léxica, podremos encontrarnos errores sintácticos: alguna expresión no tiene la forma requerida en el contexto en el que se encuentra. En estos casos el compilador no podrá seguir pues muy probablemente estos errores provoquen que todo lo que hay a continuación sea incorrecto y arreglando el primero se podrían solucionar los siguientes. Se notificará, por tanto, la línea y columna donde se ha encontrado dicho error, aunque suele ser habitual que este venga de otro lugar distinto al de su detección.

Por último, con el árbol ya construido, se procederá a la traducción. Aquí podemos encontrarnos con errores de tipos principalmente, es decir que el tipo que se esperaba para una expresión no fuera el que se ha encontrado. En este caso se notificará, al menos, el primer error encontrado.

Si todas estas fases han terminado con éxito tendremos nuestro programa compilado y listo para ser ejecutado.