Guía de Lenguaje de Programacion CompiScript

Gabriel Brolo, Bidkar Pojoy

Semestre 2, 2024

Índice

1. Introduccion **2**

2. Un lenguaje de alto nivel **2**
2.1. Gramática del lenguage . . . 3
2.2. Aspectos léxicos . . . 4
2.3. Gramática en ANTLR . . . 4

3. Tipos de Datos **5**
3.1. Booleanos . . . 5
3.2. Números . . . 5
3.3. Cadenas . . . 5
3.4. Nil . . . 6

4. Operadores y Tipos de Datos **6**
4.1. Operadores Aritméticos . . . 6
4.2. Operadores de Comparación . . . 6
4.3. Operadores Lógicos . . . 6
4.4. Precedencia y Agrupación . . . 6

5. Expresiones **7**
5.1. Literales . . . 7
5.2. Expresiones de Agrupación . . . 7
5.3. Expresiones Unarias . . . 7
5.4. Expresiones Binarias . . . 7
5.5. Expresiones de Llamada . . . 7
5.6. Expresiones de Acceso a Propiedades . . . 8
5.7. Precedencia y Asociatividad . . . 8
5.8. Aritméticas . . . 8
5.9. Comparaciones . . . 8
5.10. Logicos . . . 8
5.11. Precedencia y agrupamiento . . . 8

6. Declaraciones **9**
6.1. Declaraciones de Expresión . . . 9
6.2. Declaraciones de Impresión . . . 9
6.3. Bloques . . . 9
6.4. Declaraciones de Control de Flujo . . . 9
6.5. Declaraciones de Variables . . . 10
6.6. Declaraciones de Funciones . . . 10

7. Variables **10**
7.1. Declaración y Asignación . . . 10
7.2. Ámbito de las Variables . . . 10
7.3. Variables Globales y Locales . . . 11
7.4. Inicialización Tardía . . . 11
7.5. Variables de Solo Lectura . . . 11

8. Control de Flujo **11**
8.1. Declaraciones `if` . . . 12
8.2. Declaraciones `while` . . . 12
8.3. Declaraciones `for` . . . 12
8.4. Declaraciones `break` y `continue` . . . 12
8.5. Declaraciones `return` . . . 13

9. Funciones **13**
9.1. Closures/cerraduras . . . 13
9.2. Captura de Variables . . . 13
9.3. Ámbito Léxico . . . 14
9.4. Almacenamiento de Variables Capturadas . . . 14

10. Ejemplos de Closures **14**
10.1. Ejemplo de Captura Tardía . . . 14
10.2. Uso Práctico de las Closures . . . 15

11. Clases **15**
11.1. Definicion de Clases . . . 15
11.2. Instanciacion de Clases . . . 16
11.3. Métodos . . . 16
11.4. Inicializadores . . . 16
11.5. Campos de Instancia . . . 16
11.6. Herencia . . . 17

12. Ejemplos de Código en Compiscript **17**
12.1. Ejemplo 1: Suma Simple . . . 17
12.2. Ejemplo 2: Bucle y Condicionales . . . 18
12.3. Ejemplo 3: Sistema de Clases y Herencia . . . 18
12.4. Ejemplo 4: Función Recursiva . . . 19

1. Introduccion

Compiscript es un lenguaje de programacion diseñado para ilustrar la construccion de intérpretes. En esta guía, describimos su sintaxis y semántica, explorando como se implementan sus características.

2. Un lenguaje de alto nivel

Compiscript es un lenguaje de programacion dinámico y de tipado débil, inspirado en Python y Java. Está diseñado para ser fácil de leer y escribir.

2.1. Gramática del lenguage

A continuación la gramática en BNF:

Listing 1: Gramatica en BNF

```
program         -> declaration* EOF ;

declaration     -> classDecl
                | funDecl
                | varDecl
                | statement ;

classDecl       -> "class" IDENTIFIER ( "<" IDENTIFIER )? "{" function*
    "}" ;
funDecl         -> "fun" function ;
varDecl         -> "var" IDENTIFIER ( "=" expression )? ";" ;

statement       -> exprStmt
                | forStmt
                | ifStmt
                | printStmt
                | returnStmt
                | whileStmt
                | block ;

exprStmt        -> expression ";" ;
forStmt         -> "for" "(" ( varDecl | exprStmt | ";" ) expression? "
   ;" expression? ")" statement ;
ifStmt          -> "if" "(" expression ")" statement ( "else" statement
    )? ;
printStmt       -> "print" expression ";" ;
returnStmt      -> "return" expression? ";" ;
whileStmt       -> "while" "(" expression ")" statement ;
block           -> "{" declaration* "}" ;

expression      -> assignment ;

assignment      -> ( call "." )? IDENTIFIER "=" assignment | logic_or ;

logic_or        -> logic_and ( "or" logic_and )* ;
logic_and       -> equality ( "and" equality )* ;
equality        -> comparison ( ( "!=" | "==" ) comparison )* ;
comparison      -> term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term            -> factor ( ( "-" | "+" ) factor )* ;
factor          -> unary ( ( "/" | "*" ) unary )* ;

unary           -> ( "!" | "-" ) unary | call ;
call            -> primary ( "(" arguments? ")" | "." IDENTIFIER )* ;
primary         -> "true" | "false" | "nil" | "this" | NUMBER | STRING
   | IDENTIFIER | "(" expression ")" | "super" "." IDENTIFIER ;

function        -> IDENTIFIER "(" parameters? ")" block ;
```

```
parameters      -> IDENTIFIER ( "," IDENTIFIER )* ;
arguments       -> expression ( "," expression )* ;
```

2.2. Aspectos léxicos

```
NUMBER          -> DIGIT+ ( "." DIGIT+ )? ;
STRING          -> "\"" <any char except "\"">* "\"" ;
IDENTIFIER      -> ALPHA ( ALPHA | DIGIT )* ;
ALPHA           -> "a" ... "z" | "A" ... "Z" | "_" ;
DIGIT           -> "0" ... "9" ;
```

2.3. Gramática en ANTLR

```
grammar Compiscript;

program         : declaration* EOF ;

declaration     : classDecl
                | funDecl
                | varDecl
                | statement ;

classDecl       : 'class' IDENTIFIER ('<' IDENTIFIER)? '{' function* '
   }' ;
funDecl         : 'fun' function ;
varDecl         : 'var' IDENTIFIER ('=' expression)? ';' ;

statement       : exprStmt
                | forStmt
                | ifStmt
                | printStmt
                | returnStmt
                | whileStmt
                | block ;

exprStmt        : expression ';' ;
forStmt         : 'for' '(' (varDecl | exprStmt | ';') expression? ';'
    expression? ')' statement ;
ifStmt          : 'if' '(' expression ')' statement ('else' statement)
   ? ;
printStmt       : 'print' expression ';' ;
returnStmt      : 'return' expression? ';' ;
whileStmt       : 'while' '(' expression ')' statement ;
block           : '{' declaration* '}' ;

expression      : assignment ;

assignment      : (call '.')? IDENTIFIER '=' assignment
                | logic_or ;
```

```
logic_or        : logic_and ('or' logic_and)* ;
logic_and       : equality ('and' equality)* ;
equality        : comparison (( '!=' | '==' ) comparison)* ;
comparison      : term (( '>' | '>=' | '<' | '<=' ) term)* ;
term            : factor (( '-' | '+' ) factor)* ;
factor          : unary (( '/' | '*' ) unary)* ;

unary           : ( '!' | '-' ) unary
                | call ;
call            : primary ( '(' arguments? ')' | '.' IDENTIFIER )* ;
primary         : 'true' | 'false' | 'nil' | 'this'
                | NUMBER | STRING | IDENTIFIER | '(' expression ')'
                | 'super' '.' IDENTIFIER ;

function        : IDENTIFIER '(' parameters? ')' block ;
parameters      : IDENTIFIER ( ',' IDENTIFIER )* ;
arguments       : expression ( ',' expression )* ;

NUMBER          : DIGIT+ ( '.' DIGIT+ )? ;
STRING          : '"' (~["\\])* '"' ;
IDENTIFIER      : ALPHA ( ALPHA | DIGIT )* ;
fragment ALPHA  : [a-zA-Z_] ;
fragment DIGIT  : [0-9] ;
WS              : [ \t\r\n]+ -> skip ;
```

3. Tipos de Datos

En compiscript, los tipos de datos básicos forman la base del lenguaje. A continuación se describen los tipos de datos disponibles y cómo se utilizan.

3.1. Booleanos

Los valores booleanos representan la lógica binaria y pueden ser `true` o `false`.

Listing 2: Ejemplos de booleanos en compiscript

```
true;  // Verdadero.
false; // Falso.
```

3.2. Números

Compiscript utiliza números de punto flotante de doble precisión para representar todos los valores numéricos.

Listing 3: Ejemplos de números en compiscript

```
1234;  // Un numero entero.
12.34; // Un numero decimal.
```

3.3. Cadenas

Las cadenas son secuencias de caracteres delimitadas por comillas dobles.

Listing 4: Ejemplos de cadenas en compiscript

```
"I␣am␣a␣string";
"";    // La cadena vacia.
"123"; // Esto es una cadena, no un numero.
```

3.4. Nil

El tipo `nil` representa la ausencia de valor, similar a `null` en otros lenguajes.

Listing 5: Ejemplo de nil en compiscript

```
nil; // Representa "sin␣valor".
```

4. Operadores y Tipos de Datos

4.1. Operadores Aritméticos

Compiscript soporta operadores aritméticos básicos como la suma, resta, multiplicación y división.

Listing 6: Operadores aritméticos en compiscript

```
1 + 2;  // Suma.
3 - 1;  // Resta.
4 * 2;  // Multiplicacion.
8 / 2;  // Division.
```

4.2. Operadores de Comparación

Los operadores de comparación se utilizan para comparar números y devolver valores booleanos.

Listing 7: Operadores de comparación en compiscript

```
3 < 5;  // Menor que.
10 >= 10;  // Mayor o igual que.
1 == 1;    // Igualdad.
"a" != "b"; // Desigualdad.
```

4.3. Operadores Lógicos

Los operadores lógicos `and`, `or` y `not` se utilizan para construir expresiones lógicas.

Listing 8: Operadores lógicos en compiscript

```
true and false; // false.
true or false;  // true.
!true;          // false.
```

4.4. Precedencia y Agrupación

Los operadores en compiscript siguen reglas de precedencia y asociatividad similares a otros lenguajes. Se pueden usar paréntesis para controlar la precedencia.

Listing 9: Ejemplo de agrupación en compiscript

```
var promedio = (min + max) / 2;
```

5. Expresiones

Las expresiones en compiscript son combinaciones de valores, variables, operadores y llamadas a funciones que el intérprete evalúa para producir un valor. A continuación se describen los tipos de expresiones disponibles y cómo se utilizan.

5.1. Literales

Los literales son valores que aparecen directamente en el código. Incluyen números, cadenas, booleanos y `nil`.

Listing 10: Ejemplos de literales en compiscript

```
1234;       // Un numero.
"Hola";     // Una cadena.
true;       // Un booleano.
nil;        // Nil.
```

5.2. Expresiones de Agrupación

Las expresiones de agrupación utilizan paréntesis para controlar el orden de evaluación.

Listing 11: Expresiones de agrupación en compiscript

```
(1 + 2) * 3; // Resultado: 9.
```

5.3. Expresiones Unarias

Las expresiones unarias operan sobre un solo operando. Los operadores unarios incluyen la negación lógica y la negación aritmética.

Listing 12: Expresiones unarias en compiscript

```
!true;  // Resultado: false.
-5;     // Negacion aritmetica.
```

5.4. Expresiones Binarias

Las expresiones binarias operan sobre dos operandos. Incluyen operadores aritméticos, de comparación y lógicos.

Listing 13: Expresiones binarias en compiscript

```
1 + 2;   // Suma.
3 > 2;   // Comparacion.
true and false; // Operador logico.
```

5.5. Expresiones de Llamada

Las expresiones de llamada ejecutan funciones. Consisten en un nombre de función seguido de paréntesis que encierran los argumentos.

Listing 14: Expresiones de llamada en compiscript

```
saludar("mundo"); // Llamada a funcion.
```

5.6. Expresiones de Acceso a Propiedades

Las expresiones de acceso a propiedades obtienen el valor de una propiedad de un objeto.

Listing 15: Acceso a propiedades en compiscript

```
persona.nombre; // Acceso a la propiedad nombre.
```

5.7. Precedencia y Asociatividad

La precedencia y la asociatividad de los operadores determinan cómo se agrupan las expresiones. La precedencia especifica qué operadores se evalúan primero, mientras que la asociatividad determina el orden de evaluación de operadores con la misma precedencia.

Listing 16: Precedencia y asociatividad en compiscript

```
1 + 2 * 3;  // Resultado: 7.
(1 + 2) * 3; // Resultado: 9.
```

5.8. Aritméticas

Compiscript incluye operadores aritméticos básicos:

```
var suma = 1 + 2;
var resta = 5 - 3;
var producto = 4 * 2;
var division = 8 / 2;
```

- Un operador aritmético es en realidad tanto un operador infijo como un operador prefijo. El operador - también puede usarse para negar un número, e.g., `-negateMe`;
- Todos estos operadores funcionan con números, y es un error pasarles cualquier otro tipo. La excepcion es el operador +, que también puede usarse para concatenar dos cadenas.

5.9. Comparaciones

Operadores para comparar valores, retornando un booleano:

```
var menor = 3 < 5; // true
var mayorIgual = 10 >= 10; // true
var igual = 1 == 1; // true
var diferente = "a" != "b"; // true
```

5.10. Logicos

Operadores logicos ‘and‘, ‘or‘ y ‘not‘:

```
var y = true and false; // false
var o = true or false; // true
var no = !true; // false
```

5.11. Precedencia y agrupamiento

Los operadores tienen precedencia y asociatividad similares a los de C. Para controlar la precedencia, se pueden usar paréntesis.

```
var promedio = (min + max) / 2;
```

6. Declaraciones

Las declaraciones en compiscript son instrucciones que realizan acciones. A continuación se describen los tipos de declaraciones disponibles y cómo se utilizan.

6.1. Declaraciones de Expresión

Una declaración de expresión evalúa una expresión y descarta el valor resultante. Se utiliza comúnmente para ejecutar llamadas a funciones.

Listing 17: Declaraciones de expresión en compiscript

```
decirHola("mundo");
```

6.2. Declaraciones de Impresión

La declaración `print` evalúa una expresión y muestra el resultado al usuario.

Listing 18: Declaraciones de impresión en compiscript

```
print "Hola, mundo!";
```

6.3. Bloques

Un bloque agrupa múltiples declaraciones entre llaves. Los bloques también definen un nuevo ámbito léxico.

Listing 19: Bloques en compiscript

```
{
  var a = "dentro del bloque";
  print a;
}
```

6.4. Declaraciones de Control de Flujo

Compiscript incluye varias estructuras de control de flujo como `if`, `while` y `for`.

Listing 20: Declaraciones de control de flujo en compiscript

```
// if
if (condicion) {
  print "Condicion verdadera";
} else {
  print "Condicion falsa";
}

// while
while (condicion) {
  print "Bucle while";
}

// for
for (var i = 0; i < 10; i = i + 1) {
  print i;
}
```

6.5. Declaraciones de Variables

Las declaraciones de variables definen nuevas variables y opcionalmente las inicializan.

Listing 21: Declaraciones de variables en compiscript

```
var nombre = "Compiscript";
var edad;
```

6.6. Declaraciones de Funciones

Las declaraciones de funciones definen nuevas funciones. Incluyen un nombre, parámetros y un cuerpo de función.

Listing 22: Declaraciones de funciones en compiscript

```
fun saludar(nombre) {
  print "Hola, " + nombre;
}
```

7. Variables

En compiscript, una variable se utiliza para almacenar y manipular datos. Las variables pueden contener diferentes tipos de datos y su valor puede cambiar a lo largo del tiempo.

7.1. Declaración y Asignación

Para declarar una variable en compiscript, utilizamos la palabra clave `var` seguida del nombre de la variable y opcionalmente su valor inicial.

Listing 23: Declaración de variables en compiscript

```
var nombre;
var edad = 25;
```

7.2. Ámbito de las Variables

El ámbito de una variable determina dónde se puede acceder a ella en el código. En compiscript, las variables tienen un ámbito de bloque, lo que significa que solo están disponibles dentro del bloque donde fueron declaradas.

Listing 24: Ámbito de las variables en compiscript

```
{
  var a = "dentro del bloque";
  print a; // Imprime: dentro del bloque
}
print a; // Error: a no esta definida
```

7.3. Variables Globales y Locales

Las variables declaradas fuera de cualquier función o bloque son globales y accesibles en cualquier parte del programa. Las variables declaradas dentro de una función son locales a esa función.

Listing 25: Variables globales y locales en compiscript

```
var globalVar = "soy␣global";

fun miFuncion() {
  var localVar = "soy␣local";
  print globalVar; // Acceso permitido
  print localVar; // Acceso permitido
}

miFuncion();
print globalVar; // Acceso permitido
print localVar; // Error: localVar no esta definida
```

7.4. Inicialización Tardía

En compiscript, una variable puede ser declarada sin un valor inicial y luego ser asignada en un momento posterior.

Listing 26: Inicialización tardía de variables en compiscript

```
var miVariable;
miVariable = "Ahora␣tengo␣un␣valor";
print miVariable; // Imprime: Ahora tengo un valor
```

7.5. Variables de Solo Lectura

Compiscript no tiene una sintaxis especial para declarar variables de solo lectura, pero podemos utilizar convenciones de nombres o funciones para asegurar que ciertas variables no sean modificadas después de su inicialización.

Listing 27: Simulación de variables de solo lectura en compiscript

```
fun obtenerConstante() {
  var constante = "No␣modificar";
  return constante;
}

var constante = obtenerConstante();
print constante; // Imprime: No modificar
// constante = "Intento␣de␣modificacion"; // Error intencional
```

8. Control de Flujo

El control de flujo en compiscript permite dirigir la ejecución del programa basándose en condiciones y repetición de bloques de código.

8.1. Declaraciones `if`

La declaración `if` ejecuta un bloque de código si la condición es verdadera. Se puede añadir una cláusula `else` para manejar el caso en que la condición sea falsa.

Listing 28: Declaración `if` en compiscript

```
// Estructura basica de if
if (condicion) {
  // Codigo si la condicion es verdadera
} else {
  // Codigo si la condicion es falsa
}
```

8.2. Declaraciones `while`

La declaración `while` repite un bloque de código mientras una condición sea verdadera.

Listing 29: Declaración `while` en compiscript

```
// Estructura basica de while
while (condicion) {
  // Codigo a ejecutar mientras la condicion sea verdadera
}
```

8.3. Declaraciones `for`

La declaración `for` proporciona una forma compacta de iterar sobre un rango de valores.

Listing 30: Declaración `for` en compiscript

```
// Estructura basica de for
for (var i = 0; i < 10; i = i + 1) {
  // Codigo a ejecutar en cada iteracion
}
```

8.4. Declaraciones `break` y `continue`

Las declaraciones `break` y `continue` alteran el flujo de los bucles.

Listing 31: Declaraciones `break` y `continue` en compiscript

```
// Uso de break
while (true) {
  if (condicion) {
    break;
  }
  // Mas codigo
}

// Uso de continue
for (var i = 0; i < 10; i = i + 1) {
  if (i % 2 == 0) {
    continue;
  }
  print i; // Solo imprime numeros impares
}
```

8.5. Declaraciones `return`

La declaración `return` finaliza la ejecución de una función y opcionalmente devuelve un valor.

Listing 32: Declaración `return` en compiscript

```
fun suma(a, b) {
  return a + b;
}
```

9. Funciones

Las funciones se definen con la palabra clave 'fun' y pueden aceptar parámetros.

```
fun saludo(nombre) {
  print "Hola, " + nombre;
}

saludo("Compiscript");
```

- Un argumento es un valor real que pasas a una funcion cuando la llamas. Por lo tanto, una llamada a funcion tiene una lista de argumentos. A veces se escucha el término *parámetro real* usado para estos.
- Un parámetro es una variable que contiene el valor del argumento dentro del cuerpo de la funcion. Así, una declaracion de funcion tiene una lista de parámetros. Otros llaman a estos *parámetros formales* o simplemente *formales*.
- El cuerpo de una funcion siempre es un bloque y se puede retornar un valir usanto el statement de `return`:

```
fun returnSum(a, b) {
  return a + b;
}
```

- Si la ejecucion llega al fin del bloque sin haber un return, implícitamente regresa `nil`.

9.1. Closures/cerraduras

Una closure es una funcion que puede capturar y retener variables del ámbito en el que fue creada. En compiscript, las closures permiten a las funciones recordar y acceder a variables fuera de su propio ámbito.

9.2. Captura de Variables

Cuando una funcion se define dentro de otra, la funcion interna tiene acceso a las variables locales de la funcion externa. En compiscript, cuando se crea una closure, esta captura las variables locales que utiliza. Aquí hay un ejemplo en compiscript:

Listing 33: Ejemplo de closure en compiscript

```
fun hacerContador() {
  var i = 0;
  fun contar() {
```

```
    i = i + 1;
    print i;
  }

  return contar;
}

var contador = hacerContador();
contador(); // "1".
contador(); // "2".
```

En este ejemplo, la funcion `contar` es una closure que captura la variable `i` de la funcion `hacerContador`.

9.3. Ámbito Léxico

El ámbito léxico se refiere al alcance de las variables basándose en la estructura del codigo fuente. En compiscript, las closures siguen las reglas del ámbito léxico, permitiendo a las funciones acceder a variables que estaban en su alcance en el momento en que fueron definidas.

9.4. Almacenamiento de Variables Capturadas

En la implementacion de un lenguaje, es crucial manejar como y donde se almacenan las variables capturadas por las closures. Una forma común es almacenar estas variables en el heap para que persistan más allá del ámbito de la funcion que las define.

Listing 34: Otro ejemplo de closure en compiscript

```
fun crearSumador(n) {
  return fun(x) {
    return x + n;
  };
}

var suma5 = crearSumador(5);
print suma5(10); // "15".
print suma5(20); // "25".
```

En este caso, la funcion anonima captura la variable `n` de la funcion `crearSumador`.

10. Ejemplos de Closures

10.1. Ejemplo de Captura Tardía

Un aspecto importante de las closures es cuándo capturan las variables. Compiscript implementa captura tardía, lo que significa que las variables se capturan cuando la closure se ejecuta, no cuando se define.

Listing 35: Ejemplo de captura tardía en compiscript

```
fun hacerFunciones() {
  var funciones = [];

  for (var i = 0; i < 3; i = i + 1) {
    fun imprimir() {
      print i;
```

```
    }
    funciones.add(imprimir);
  }

  return funciones;
}

var misFunciones = hacerFunciones();
misFunciones[0](); // "3".
misFunciones[1](); // "3".
misFunciones[2](); // "3".
```

Aquí, todas las funciones capturan la misma variable `i`, que tiene el valor 3 al final del bucle.

10.2. Uso Práctico de las Closures

Las closures son útiles para crear funciones con estado privado o para definir funciones de orden superior que pueden personalizarse con variables de su ámbito externo.

Listing 36: Ejemplo práctico de closure en compiscript

```
fun crearContador() {
  var contador = 0;
  return fun() {
    contador = contador + 1;
    return contador;
  };
}

var contar = crearContador();
print contar(); // "1".
print contar(); // "2".
```

En este ejemplo, la funcion `contar` mantiene el estado del contador entre invocaciones sucesivas.

11. Clases

Las clases en Compiscript permiten definir tipos complejos que encapsulan datos y comportamientos. A continuacion se detalla como funcionan las clases en Compiscript.

11.1. Definicion de Clases

Una clase se define con la palabra clave `class` seguida del nombre de la clase. El cuerpo de la clase contiene los métodos que definen su comportamiento.

Listing 37: Definicion de una clase en Compiscript

```
class Persona {
  decirHola() {
    print "Hola, mundo!";
  }
}
```

11.2. Instanciacion de Clases

Para crear una instancia de una clase, se utiliza el operador `new`.

Listing 38: Creacion de una instancia de la clase

```
var juan = new Persona();
juan.decirHola(); // Salida: Hola, mundo!
```

11.3. Métodos

Los métodos son funciones que pertenecen a una clase. En Compiscript, los métodos se definen dentro del cuerpo de la clase.

Listing 39: Definicion de un método en una clase

```
class Persona {
  decirHola() {
    print "Hola, mundo!";
  }
}
```

11.4. Inicializadores

El inicializador es un método especial llamado `init`, que se llama automáticamente cuando se crea una instancia de la clase.

Listing 40: Uso de un inicializador

```
class Persona {
  init(nombre) {
    this.nombre = nombre;
  }

  decirNombre() {
    print this.nombre;
  }
}

var juan = new Persona("Juan");
juan.decirNombre(); // Salida: Juan
```

11.5. Campos de Instancia

Los campos de instancia son variables que pertenecen a una instancia específica de una clase. Se pueden inicializar en el método `init`.

Listing 41: Definicion de campos de instancia

```
class Persona {
  init(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
```

```
  presentar() {
    print this.nombre + "␣tiene␣" + this.edad + "␣anios.";
  }
}

var maria = new Persona("Maria", 30);
maria.presentar(); // Salida: Maria tiene 30 anios.
```

11.6. Herencia

Compiscript soporta herencia, lo que permite crear nuevas clases basadas en clases existentes.

Listing 42: Herencia en Compiscript

```
class Persona {
  init(nombre) {
    this.nombre = nombre;
  }

  decirNombre() {
    print this.nombre;
  }
}

class Estudiante extends Persona {
  init(nombre, grado) {
    super.init(nombre);
    this.grado = grado;
  }

  decirGrado() {
    print this.nombre + "␣esta␣en␣" + this.grado + ".";
  }
}

var ana = new Estudiante("Ana", "tercer␣grado");
ana.decirGrado(); // Salida: Ana esta en tercer grado.
```

12. Ejemplos de Código en Compiscript

12.1. Ejemplo 1: Suma Simple

Listing 43: Suma Simple

```
fun suma(a, b) {
  return a + b;
}

print suma(3, 4); // Salida: 7
```

12.2. Ejemplo 2: Bucle y Condicionales

Listing 44: Bucle y Condicionales

```
fun esPar(num) {
  return num % 2 == 0;
}

for (var i = 1; i <= 10; i = i + 1) {
  if (esPar(i)) {
    print i + "␣es␣par";
  } else {
    print i + "␣es␣impar";
  }
}
```

12.3. Ejemplo 3: Sistema de Clases y Herencia

Listing 45: Sistema de Clases y Herencia

```
class Persona {
  init(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  saludar() {
    print "Hola,␣mi␣nombre␣es␣" + this.nombre;
  }
}

class Estudiante extends Persona {
  init(nombre, edad, grado) {
    super.init(nombre, edad);
    this.grado = grado;
  }

  estudiar() {
    print this.nombre + "␣esta␣estudiando␣en␣" + this.grado + "␣grado.
       ";
  }
}

var juan = Estudiante("Juan", 20, 3);
juan.saludar();    // Salida: Hola, mi nombre es Juan
juan.estudiar();   // Salida: Juan esta estudiando en 3 grado

for (var i = 1; i <= 5; i = i + 1) {
  if (i % 2 == 0) {
    print i + "␣es␣par";
  } else {
    print i + "␣es␣impar";
```

```
  }
}

while (juan.edad < 25) {
  juan.edad = juan.edad + 1;
  print "Edad de Juan: " + juan.edad;
}
```

12.4. Ejemplo 4: Función Recursiva

Listing 46: Función Recursiva

```
fun factorial(n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}

print "Factorial de 5: " + factorial(5); // Salida: Factorial de 5:
    120

fun fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

print "Fibonacci de 10: " + fibonacci(10); // Salida: Fibonacci de 10:
     55
```