



Tecnológico Nacional de México.

Instituto Tecnológico de Veracruz

Materia: Lenguajes y Autómatas II.

Grupo: 7j1.

Hora: 12:00 pm – 13:00 pm.

Alumnos: Vilma Zorina Camacho Cágal.

José Manuel Miranda Villagrán.

Periodo: Agosto–Diciembre.

Docente: Martínez Moreno Martha.

Índice

Introducción	8
Antecedentes del lenguaje	9
Simbología	10
Palabras Reservadas	10
Operadores Aritméticos	13
Operadores Relacionales	14
Operadores Lógicos	15
Expresiones regulares	16
Gramática	22
Sintaxis del lenguaje	22
Ejemplos sencillos	22
El bucle de entrada/evaluación	25
Cadenas	26
Expresiones regulares	30
Arrays	34
Hashes	36
Retomando Los Ejemplos Sencillos	37
Factoriales	37
Estructuras de control	39
Case	39
while	40
For	42
Iteradores	44
Métodos	48
Clases	51
Herencia	53
Redefinición de métodos	55
Control de accesos	58
Módulos	61
Variables	62

Variables Globales	63
Variables de Instancia	66
Variables Locales	67
Constantes	70
Fuente jj	73
Semántica	83
Descripción	83
Tratamiento de errores semánticos	83
Error de parseo	83
División entre Cero	85
División entre distintos tipos de datos	87
Sensible a las mayúsculas	88
Ingresar un Dato Erróneo	90
Multiplicar Cadenas	92
Función de Parámetros Faltantes	94
Arreglos con el mismo nombre	95
Variable no Inicializada	97
suma usando un string y un entero	99
Fuente jj	101
Implementación de un error semántico	103
Instalación	104
Requerimientos	104
Step by step	104
Entorno de edición	109
Vista general del compilador MCvill Compiler.	109
Menú "Archivo"	110
Abrir	110
Guardar	111
Salir	112
Menú "Acciones"	113
Compilar	113
Limpiar	114
Contador de Líneas y Panel Central para el Código Fuente	115
Ventanas de Análisis	115
Ventana del Análisis Léxico	116

Ventana del Análisis Sintáctico	117
Ventana del Análisis Semántico	118
Compilación	119
Sin error	119
Con error	120
Bibliografía	121

Índice de Tablas

Tabla 1 - Palabras Reservadas	12
Tabla 2 - Operadores Aritméticos	13
Tabla 3 - Operadores Relacionales	14
Tabla 4 - Operadores Lógicos	15
Tabla 5 - Expresiones Regulares	31
Tabla 6 - Variables	63
Tabla 7 - Variables del Sistema	65
Tabla 8 - Requerimientos de Instalación	104

Índice de Figuras

Figura 1 - ER para ignorar espacios, tabulaciones, y retornos de carro	16
Figura 2 - ER para comentarios multilínea	16
Figura 3 - ER para palabras reservadas del lenguaje parte 1.	17
Figura 4 - ER para palabras reservadas del lenguaje parte 2.	18
Figura 5 - ER para palabras reservadas del lenguaje parte 3.	19
Figura 6 - ER para delimitadores y símbolos qué tienen más funciones.	19
Figura 7 - ER para operadores aritméticos.	20
Figura 8 - ER para operadores relacionales.	20
Figura 9 - ER para operadores de asignación aritmética.	20
Figura 10 - ER para símbolos especiales de Ruby.	21
Figura 10 - ER para símbolos especiales de Ruby.	21
Figura 12 - ER para definir tipos y valores de Ruby.	21
Figura 13 - Uso de la función gets.	83
Figura 14.- Compilación sin Parseo (Error de Conversión).	84
Figura 15.- Uso de la función .to_i.	84
Figura 16 - Compilación con Parseo.	85
Figura 17.- Compilación división entre cero.	86
Figura 18.- Parseo a números enteros.	87
Figura 19.- Compilación división de dos números flotantes.	87
Figura 20 - Opciones declaradas en minúsculas.	88
Figura 21.- Compilación ingresando una letra mayúscula.	89

Figura 22 - Opciones declaradas en minúsculas.	90
Figura 23 - Compilación ingresando un número.	91
Figura 24 - Uso del símbolo * en la impresión.	92
Figura 25 -Compilación Impresión múltiple de la cadena.	93
Figura 26 - Llamada al método multiplica.	94
Figura 27 - Compilación ingresando una letra mayúscula.	94
Figura 28 - Duplicidad del arreglo vec2.	95
Figura 29 - Compilación de la impresión del arreglo duplicado vec2.	95
Figura 30 - Declaración de tres arreglos.	96
Figura 31 - Compilación de la impresión del arreglo vec2.	96
Figura 32 - Variable num no inicializada	97
Figura 33 - Compilación de la asignación de la variable num al arreglo vec.	97
Figura 34 - Declaración y asignación de la variable num.	98
Figura 35 - Compilación impresión del arreglo vec.	98
Figura 36 - Suma usando una cadena y un valor numérico.	99
Figura 37 - Compilación de la suma usando una cadena y un valor numérico.	99
Figura 38 - suma de dos cadenas.	100
Figura 39 - Compilación de la suma usando dos cadenas.	100
Figura 40 - Acciones semánticas para almacenar tipos de valores.	101
Figura 41 - Acción semántica para añadir nuevas variables...	102
Figura 42 - Algoritmo para añadir pares ordenados...	103
Figura 43 - Instalador de MCVill Compiler.	104
Figura 44 - Acuerdo de Licencia	105
Figura 45 - Información "Requisitos del Sistema"	105
Figura 46 - Tareas Adicionales	106

Figura 47 - Panorama General de la Instalación.	106
Figura 48 - Información de Instalación de MCVill Compiler.	107
Figura 49 - Instalación Completa.	107
Figura 50 - Ejecutando MCVill Compiler.	108
Figura 51 - Vista Previa de MCVill Compiler.	108
Figura 52 - Vista General de MCVill Compiler.	109
Figura 53 - Menú Archivo	110
Figura 54 - Submenú "Abrir"	110
Figura 55 - Ventana Emergente "Abrir Archivo"	111
Figura 56 - Submenú "Guardar"	111
Figura 57 - Ventana Emergente "Guardar"	112
Figura 58 - Submenú "Salir"	112
Figura 59 - Menú Acciones	113
Figura 60 - Submenú "Compilar"	113
Figura 61 - Vista General al realizar la acción "Compilar"	114
Figura 62 - Submenú "Limpiar"	114
Figura 63 - Contador de Líneas y Panel Central	115
Figura 64 - Ventanas de Análisis	115
Figura 65 - Ventana del Análisis Léxico	116
Figura 66 - Ventana del Análisis Sintáctico	117
Figura 67 - Ventana del Análisis Semántico	118
Figura 68 - Compilación sin errores.	119
Figura 69 - Compilación con errores.	120

Introducción

Actualmente, existe una gran cantidad de lenguajes de programación que son utilizados en diversas áreas, desde el desarrollo de aplicaciones administrativas hasta el campo de la inteligencia artificial.

Es muy complicado conocer a fondo las posibilidades que nos presenta cada uno de estos lenguajes y, por lo tanto, a la hora de seleccionar, lo hacemos sobre la base de nuestros gustos o inquietudes. Ruby se presenta como un lenguaje **sencillo y flexible** que atrae a programadores de todos los sectores y que promete una grata experiencia en el trabajo habitual.

A pesar de tener muchos años en el mercado, el auge del lenguaje llegó de la mano de un framework para aplicaciones web denominado **Rails**. Esto hizo que muchos desarrolladores web migrarán desde sus lenguajes más tradicionales, como **PHP** o **ASP**, a la nueva y fascinante opción. Sin embargo, Ruby es un lenguaje multipropósito que permite desarrollos en las siguientes áreas:

- aplicaciones comerciales.
- acceso a base de datos.
- proceso y transformación de XML.
- aplicaciones distribuidas.
- aplicaciones web.

Antecedentes del lenguaje

Ruby fue creado en el Japón por Yukihiro Matsumoto mientras trabajaba como programador con lenguajes como Perl y PHP. En principio, su intención fue la de crear un Perl avanzado debido a que deseaba mejorar algunas de las apreciadas particularidades de este conocido lenguaje. Pero en lugar de mejorarlo, se vio tentado a desarrollar uno propio a partir de sus lenguajes preferidos: Perl, Smalltalk, Eiffel y Lisp. De esta forma surge el lenguaje Ruby, aunque en ese momento aún no contaba con ninguna línea de código.

Luego de más de dos años de trabajo, Ruby se presenta al público en su versión 0.95. En esta etapa, todo lo relacionado con el lenguaje era precario y todavía no contaba con gran empuje; tanto es así que se anuncia que el CVS sería lanzado semanas después. Finalmente, en 1996, Ruby 1.0 es ofrecido al público. A partir de 1997, varias empresas se interesan en Ruby como un campo para explorar, y ese mismo año se escribe el primer artículo técnico.

Un año después, aparece la página oficial en idioma inglés; empiezan a hacerse charlas y conferencias sobre el lenguaje, con gran aceptación en los ambientes académicos. En el año 2000, IBM se interesa en el lenguaje y publica un artículo acerca de la denominada *Latest open source gem from Japan* (La última gema del open source del Japón). El lenguaje creció de forma lenta, pero sostenida, hasta el 2004, cuando Rails fue liberado. David Heinemeier Hansson crea este framework cuya primera versión (1.0) salió definitivamente un año después. A partir de la aparición de Rails, el crecimiento de Ruby ha sido extraordinario: se lo ha seleccionado como **el lenguaje de programación del 2006** y se encuentra entre los 10 más populares de la actualidad según el ranking **TIOBE**.

Simbología

Palabras Reservadas

Como todo lenguaje de programación, Ruby tiene una lista de palabras que están reservadas (es decir, que no pueden ser invocadas como nombre de una variable, por ejemplo) para su uso exclusivo. La siguiente tabla muestra las palabras que caen en dicha categoría, dando una breve explicación de cada una.

Valor	Tipo
alias	Crea un alias para un operador, método o variable global que ya exista.
and	Operador lógico, igual a <code>&&</code> pero con menor precedencia.
break	Finaliza un <i>while</i> o un <i>until loop</i> , o un método dentro de un bloque
case	Compara una expresión con una clausula <i>when</i> correspondiente
class	Define una clase; se cierra con <i>end</i> .
def	Inicia la definición de un método; se cierra con <i>end</i> .
defined?	Determina si un método, una variable o un bloque existe.
do	Comienza un bloque; se cierra con <i>end</i> .
else	Ejecuta el código que continua si la condición previa no es <i>true</i> . Funciona con <i>if</i> , <i>elsif</i> , <i>unless</i> o <i>case</i> .

elsif	Ejecuta el código que continua si la condicional previa no es <i>true</i> . Funciona con <i>if</i> o <i>elsif</i> .
end	Finaliza un bloque de código.
false	Lógico o Booleano <i>false</i> .
true	Lógico o Booleano <i>true</i> .
for	Comienza un loop <i>for</i> . Se usa con <i>in</i> .
if	Ejecuta un bloque de código si la declaración condicional es <i>true</i> . Se cierra con <i>end</i> .
in	Usado con el loop <i>for</i> .
module	Define un módulo. Se cierra con <i>end</i> .
next	Salta al punto inmediatamente después de la evaluación del loop condicional
not	Operador lógico, igual como !
or	Operador lógico, igual a // pero con menor precedencia.
rescue	Evalúa una expresión después de una excepción es alzada. Usada después de <i>ensure</i> .
retry	Cuando es llamada fuera de <i>rescue</i> , repite una llamada a método. Dentro de <i>rescue</i> salta a un bloque superior.
return	Regresa un valor de un método o un bloque.
Self	Objeto contemporáneo. Alude al objeto mismo.
super	Llamada a método del mismo nombre en la superclase.

then	Separador usado con <i>if</i> , <i>unless</i> , <i>when</i> , <i>case</i> , y <i>rescue</i> .
undef	Crea un método indefinido en la clase contemporánea.
unless	Ejecuta un bloque de código si la declaración condicional es <i>false</i> .
until	Ejecuta un bloque de código mientras la declaración condicional es <i>false</i> .
when	Inicia una clausula debajo de <i>under</i> .
while	Ejecuta un bloque de código mientras la declaración condicional es <i>true</i> .
yield	Ejecuta un bloque pasado a un método.
FILE	Nombre del archivo de origen contemporáneo.
LINE	Número de la línea contemporánea en el archivo de origen contemporáneo.
puts // print	Sentencia para mostrar información en pantalla

Tabla 1 .- Palabras Reservadas

Operadores Aritméticos

Los operadores aritméticos permiten realizar operaciones matemáticas, entre dos variables y/o constantes. Todas las expresiones entre paréntesis se evalúan primero. Las expresiones con paréntesis anidados se evalúan de adentro hacia afuera, el paréntesis más interno se evalúa primero:

Operador	Descripción	Ejemplo	Resultado
+	Suma	$4 + 9$	13
-	Resta	$5 - 2$	3
*	Multiplicación	$3 * 8$	24
**	Potencia	$4 ** 2$	16
/	División	$8.4 / 2$	4.2
%	Módulo	$8 \% 3$	2

Tabla 2 .- Operadores Aritméticos

Operadores Relacionales

Los operadores relacionales comparan valores entre sí, el resultado es verdadero o false (uno o cero).

Operador	Descripción
==	igual que
!=	diferente de
<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que

Tabla 3 - Operadores Relacionales

Operadores Lógicos

Los operadores lógicos se utilizan para comparar dos expresiones y devolver un resultado booleano (verdadero o falso). Estos operadores unen estas expresiones devolviendo también verdadero o falso.

Operador	Descripción
&&, and	Y
, or	O
!	No

Tabla 4 - Operadores Lógicos

Expresiones regulares

A continuación se mostrarán capturas del archivo .jj de las expresiones regulares que forman parte del análisis lexicográfico.

```
//IGNORAR
SKIP : {
    | " "
    | "\t"
    | "\r"
    | "<"rem" (~["\n","\r"])* (" \n" | " \r" | " \r\n")>
}
```

Figura 1 - ER para ignorar espacios, tabulaciones, y retornos de carro.

```
//COMENTARIOS
SKIP : {
    | "=begin" : COMENTARIO
    | "#" : COMENTARIO
}
<COMENTARIO> SKIP : {
    | "=end" : DEFAULT
    | "\n" : DEFAULT
}
<COMENTARIO> MORE : {
    | <~[]>
}
```

Figura 2 - ER para comentarios multilínea.

```
//SIMBOLOGIA
TOKEN[IGNORE_CASE]:{

    <PR_ALIAS:"alias">
    <PR_AND:"and">
    <PR_BREAK:"break">
    <PR_CASE:"case">
    <PR_CLASS:"class">
    <PR_DEF:"def">
    <PR_DEFINED:"defined">
    <PR_DO:"do">
    <PR_ELSE:"else">
    <PR_ELSEIF:"elsif">
    <PR_END:"end">
    <PR_ENSURE:"ensure">
    <PR_FALSE:"false">
    <PR_TRUE:"true">
    <PR_FOR:"for">
    <PR_IF:"if">
    <PR_IN:"in">
    <PR_MODULE:"module">
    <PR_NEXT:"next">
    <PR_NIL:"nil">
    <PR_NOT:"not">
    <PR_OR:"or">
```

Figura 3 - ER para palabras reservadas del lenguaje parte 1.

```
<PR_OR:"or">
<PR_REDO:"redo">
<PR_RESCUE:"rescue">
<PR_RETRY:"retry">
<PR_RETURN:"return">
<PR_SELF:"self">
<PR_SUPER:"super">
<PR_THEN:"then">
<PR_UNDEF:"undef">
<PR_UNLESS:"unless">
<PR_UNTIL:"until">
<PR_WHEN:"when">
<PR_WHILE:"while">
<PR_YIELD:"yield">
<PR_FILE:"_FILE_">
<PR_LINE:"_LINE_">
<PR_PRINT:"print">
<PR_PRINTF:"printf">
<PR_PUTS:"puts">
<PR_GETS:"gets">
<PR_CHOMP:"chomp">
<PR_TO_I:"to_i">
<PR_TO_S:"to_s">
<PR_TO_F:"to_f">
```

Figura 4 - ER para palabras reservadas del lenguaje parte 2.

```

| <PR_NEW:"new">
| <PR_P:"p">
| <PR_EACH:"each">
| <PR_EMPTY:"empty">
| <PR_UPCASE:"upcase">
| <PR_INCLUDE:"include">
}

```

Figura 5 - ER para palabras reservadas del lenguaje parte 3.

```

//SIMBOLOS Y OPERADORES
TOKEN:{

```

```

| <S_SUMA:"+">
| <S_MULTIPLICA:"*">
| <S_LLAVE_A:"{">
| <S_LLAVE_C:"}">
| <S_PUNTO:".">
| <S_CORCHETE_A:"[">
| <S_CORCHETE_C:"]">
| <S_PAREN_A:"(">
| <S_PAREN_C:")">
| <S_COMA:", ">
| <S_PUNTO_COMA:";">
| <S_PUNTOS:" ":">
| <S_BARRA:"|">

```

Figura 6 - ER para delimitadores y símbolos que tienen más funciones.

```
<OA_RESTA: "-">  
<OA_DIVISION: "/">  
<OA_POTENCIA: "**">  
<OA_MOD_DIV: "%">
```

Figura 7 - ER para operadores aritméticos.

```
<OC_IGUAL_QUE: "==">  
<OC_DIFERENTE_DE: "!=">  
<OC_MENOR_QUE: "<">  
<OC_MAYOR_QUE: ">">  
<OC_MENOR_O_IGUAL: "<=">  
<OC_MAYOR_O_IGUAL: ">=">  
<OC_IGUALDAD_WHEN: "===">  
<OC_COMBINADO: "<=>">  
<OC_QUESTION: "?">
```

Figura 8 - ER para operadores relacionales.

```
<OAS_ASIGNA: "=">  
<OAS_SUMA_ASIGNA: "+=">  
<OAS_RESTA_ASIGNA: "-=">  
<OAS_MULTIPLICACION_ASIGNA: "*=">  
<OAS_DIVISION_ASIGNA: "/=">  
<OC_MODULO_ASIGNA: "%=">  
<OC_POTENCIA_ASIGNA: "**=">
```

Figura 9 - ER para operadores de asignación aritmética.


```

| <ORA_RANGO: ". ." >
| <ORA_RANGO1: ". . ." >
| <OACC_CONSTANTES: " :: " >

```

Figura 10 - ER para símbolos especiales de Ruby.

```

| <OL_AND: "&&" >
| <OL_OR: " || " >
| <OL_NOT: " ! " >
}

```

Figura 11 - ER para operadores lógicos

```

//VALORES Y TIPOS
TOKEN:{
    <VAL_CADENA: ( "\"" (~["\""]) * "\"" ) >
    <VAL_CADENA2: ( "'" (~["'"]) * "'" ) >
    <VAL_ENTERO: ("-"?) ([ "0" - "9" ])+ >
    <VAL_DECIMAL: ("-"?) (<VAL_ENTERO>)* ( "." ) (<VAL_ENTERO>)+ >
    <ID_LOCAL: ("_"?) ([ "a" - "z", "A" - "Z" ])( [ "a" - "z", "A" - "Z", "0" - "9", "_" ])* >
    <ID_INSTANCE: ("@" ) (<ID_LOCAL>)+ >
    <ID_CLASS: ("@"@" ) (<ID_LOCAL>)+ >
    <ID_GLOBAL: (" $" ) (<ID_LOCAL>)+ >
    <ID_BLOCK: ("&" ) (<ID_LOCAL>)+ >
    <ID_CONSTANT: ( [ "A" - "Z" ])+ >
    <S_SALTO: ( "\n" )+ >
}

```

Figura 12 - ER para definir tipos y valores de Ruby.

Gramática

Sintaxis del lenguaje

Ejemplos sencillos

Escribamos una función que calcula factoriales. La definición matemática de factorial es la siguiente:

(n==0) $n! = 1$

(sino) $n! = n * (n-1)!$

En Ruby se puede escribir así:

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
```

Se puede apreciar la aparición repetida de end. Debido a esto a Ruby se le conoce como un lenguaje “tipo Algol”. (Realmente la sintaxis de Ruby reproduce con más exactitud la del lenguaje Eiffel). También se puede apreciar la falta de la sentencia return. Es innecesaria debido a que una función Ruby devuelve lo último que haya evaluado.

La utilización de return es factible aunque innecesaria.

Probemos la función factorial. Añadiendo una línea de código obtenemos un programa funcional:

```
# Programa para hallar el factorial de un número
```

```
# Guarda este programa como fact.rb
```

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end

print fact(ARGV[0].to_i), "\n"
```

Aquí, ARGV es un array que contiene los parámetros de la línea de comandos y to_i convierte una cadena de caracteres a un entero.

```
% ruby fact.rb 1
```

```
1
```

```
% ruby fact.rb 5
```

```
120
```


¿Funciona con un parámetro igual a 40? Este valor podría provocar un desbordamiento en una calculadora...

```
% ruby fact.rb 40
```

```
815915283247897734345611269596115894272000000000
```

Funciona. Además Ruby puede tratar cualquier entero que quepa en la memoria del ordenador. ¡Por lo que se puede calcular el factorial de 400!:

```
% ruby fact.rb 400
```

```
6403452284662389526234797031950300585070258302600295945868444594280
2397169186831436278478647463264676294350575035856810848298162883517
4352289619886468029979373416541508381624264619423523070462443250151
1444867089066277391491811733195599644070954967134529047702032243491
1210797593280795101545372667251627877890009349763765710326350331533
9653498683868313393520243737881577867915063118587026182701698197400
6298302530859129834616227230455833952075961150530223608681043329725
5194852674432232438669948422404232599805551610635942376961399231917
1340638589965379701478272066063202173794720103213566246138090779423
0459736069956759583609615871512991382228657857954936161765448045322
2007825818400848436415591229454275384803558374518022675900061399560
14559520612721119291810503249100800000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
```

El bucle de entrada/evaluación

Al ejecutar Ruby sin parámetros, éste lee de la entrada estándar comandos y los ejecuta después de dar por finalizada la entrada:

```
% ruby
```

```
print "hola mundo\n"
```

```
print "adiós mundo\n"
```

```
^D
```

```
hola mundo
```

```
adiós mundo
```

Ruby incluye un programa llamado **eval.rb** que permite la introducción de código desde el teclado a través de un bucle iterativo que muestra los resultados a medida que se obtienen. Se utilizará ampliamente a lo largo del tutorial.

Si se tiene un terminal ANSI (esto será casi seguro si se está ejecutando alguna versión de UNIX; bajo DOS se debe instalar **ANSI.SYS** o **ANSI.COM**) se debería utilizar este **eval.rb** mejorado que permite autoindentación, informes sobre incidencias y resaltado por color. Si no, en el directorio **sample** de la distribución Ruby existe una versión no ANSI que funciona con cualquier terminal. A continuación se muestra una pequeña sesión con **eval.rb**:

```
%ruby eval.rb
```

```
ruby> print "hola mundo\n"
```

```
hola mundo.
```

```
    nil
```

```
ruby> exit
```

print produce *hola mundo*. La siguiente línea, es este caso *nil* informa sobre lo último que se ha evaluado; Ruby no distingue entre *sentencias* y *expresiones*, por lo tanto la evaluación de una pieza de código significa básicamente lo mismo que ejecutarla. Aquí *nil*, indica que *print* no devuelve ningún valor significativo. Obsérvese que se puede salir del bucle de interpretación con *exit*, aunque también funciona *^D*.

A lo largo de toda esta introducción **ruby>** representa la petición de entrada del pequeño pero útil programa **eval.rb**.

Cadenas

Ruby maneja tanto cadenas como datos numéricos. Las cadenas pueden estar entre comillas dobles("...") o comillas simples ('...').

```
ruby> "abc"
```

```
"abc"
```

```
ruby> 'abc'
```

```
"abc"
```

Las comillas simples y dobles a veces tienen efectos diferentes. Una cadena de comillas dobles permite la presencia embebida de caracteres de escape precedidos por un backslash y la expresión de evaluación `#{ }`.

Una cadena de comillas simples no realiza esta evaluación, lo que se ve es lo que se obtiene. Ejemplos:

```
ruby> print "a\nb\nc","\n"
```

a

b

c

nil

```
ruby> print 'a\nb\nc','\n'
```

a\nb\nc

nil

```
ruby> "\n"
```

"\n"

```
ruby> '\n'
```

"\n"

```
ruby> "\001"
```

"\001"

```
ruby> '\001'
```

"\001"

```
ruby> "abcd #{5*3} efg"
```

"abcd 15 efg"

```
ruby> var = " abc "
```

" abc "

```
ruby> "1234#{var}5678"
```

"1234 abc 5678"

El manejo de las cadenas en Ruby es más inteligente e intuitivo que en C. Por ejemplo, se pueden concatenar cadenas con + y se puede repetir una cadena varias veces con *:

```
ruby> "foo" + "bar"
```

```
"foobar"
```

```
ruby> "foo" * 2
```

```
"foofoo"
```

La concatenación de cadenas en C es más incómoda debido a la necesidad de una gestión explícita de la memoria:

```
char *s = malloc(strlen(s1)+strlen(s2) +1);
```

```
strcpy(s, s1);
```

```
strcat(s, s2);
```

```
/* ... */
```

```
free(s);
```

Ya que al usar Ruby no tenemos que considerar el espacio que va a ocupar una cadena, estamos liberados de la gestión de la memoria.

A continuación se muestran algunas cosas que se pueden hacer con las cadenas.
Concatenación:

```
ruby> word = "fo" + "o"
```

```
"foo"
```

Repetición:

```
ruby> word = word * 2
```

```
"foofoo"
```

Extracción de caracteres (obsérvese que los caracteres en Ruby son enteros):

```
ruby> word[0]
```

```
102          # 102 es el código ASCII de 'f'
```

```
ruby> word[-1]
```

```
111          # 111 es el código ASCII de 'o'
```

(Los índices negativos indican desplazamientos desde el final de la cadena, en vez del comienzo).

Extracción de subcadenas:

```
ruby> herb = "parsley"
```

```
"parsley"
```

```
ruby> herb[0,1]
```

```
"p"
```

```
ruby> herb[-2,2]
```

```
"ey"
```

```
ruby> herb[0..3]
```

```
"pars"
```

```
ruby> herb[-5..-2]
```

```
"rsle"
```

Verificación de la igualdad:

```
ruby> "foo" == "foo"
```

```
true
```

```
ruby> "foo" == "bar"
```

```
false
```

Expresiones regulares

Realicemos un programa mucho más interesante. Es el momento de comprobar si una cadena satisface una descripción, que llamaremos patrón.

En estos patrones existen algunos caracteres y combinaciones de caracteres que tienen un significado especial, y son:

Símbolo	Descripción
[]	Especificación de Rango. (p.e. [a-z] representa una letra en el rango de la a a la z).
\w	Letra o Dígito; es lo mismo que [0-9 A-Z a-z].
\W	Ni letra, ni dígito.
\s	Espacio, es lo mismo que [\t \n \r \f]
\S	No espacio.
\d	Dígito; es lo mismo que [0-9]
\D	No digitio
\b	Backspace (0x80) (Sólo si aparece en una especificación de rango)
\b	Límite de palabra (sólo si no aparece en una especificación de rango)
\B	No límite de palabra
*	cero o más repeticiones de lo que precede
+	Una o más repeticiones de lo que precede
[m,n]	Al menos <i>m</i> y como máximo <i>n</i> de lo que

	precede
?	Al menos una repetición de lo que precede; es lo mismo que [0,1]
	Puede coincidir con lo que precede o con lo que sigue.
()	Agrupamiento

Tabla 5.- Expresiones Regulares

El término común para estos patrones que utilizan este extraño vocabulario es *expresión regular*. En Ruby, como en Perl, normalmente están rodeadas por barras inclinadas en vez de por comillas dobles. Si nunca antes se ha trabajado con expresiones regulares, es probable que parecen cualquier cosa excepto regulares, pero sería inteligente dedicar algún tiempo a familiarizarse con ellas. Tienen un poder expresivo en su concisión que puede evitar muchos dolores de cabeza (y muchas líneas de código) si se necesita realizar coincidencia de patrones o cualquier otro tipo de manipulación con cadenas de texto.

Por ejemplo, supongamos que queremos comprobar si una cadena se ajusta a esta descripción: "Comienza con una f minúscula, a la que sigue exactamente una letra mayúscula y opcionalmente cualquier cosa detrás de ésta, siempre y cuando no haya más letras minúsculas." Si se es un experimentado programador en C probablemente se haya escrito este tipo de código docenas de veces, ¿verdad? Admitámoslo, es difícil mejorarlo. Pero en Ruby solamente es necesario solicitar que se verifique la cadena contra la siguiente expresión regular `/^f[A-Z][^a-z]*$/`.

Y qué decir de "¿Contiene la cadena un número hexadecimal entre ángulos?" No hay problema.


```
ruby> def chab(s) # contiene la cadena un hexadecimal entre ángulos
```

```
ruby|      (s =~ /<0[Xx][\dA-Fa-f]+>/) != nil
```

```
ruby| end
```

```
nil
```

```
ruby> chab "Este no es"
```

```
false
```

```
ruby> chab "¿Puede ser esta? (0x35)" # entre paréntesis, no ángulos
```

```
false
```

```
ruby> chab "¿O esta? <0x38z7e>" # letra errónea
```

```
false
```

```
ruby> chab "OK esta si; <0xfc0004>"
```

```
true
```

Aunque inicialmente las expresiones regulares pueden parecer enigmáticas, se gana rápidamente satisfacción al ser capaz de expresarse con tanta economía.

A continuación se presenta un pequeño programa que nos permitirá experimentar con las expresiones regulares. Almacenemos como **regx.rb**, se ejecuta introduciendo en la línea de comandos ruby **regx.rb**

```
# necesita un terminal ANSI
```

```
st = "\033[7m"
```

```
en = "\033[m"
```

```
while TRUE
```

```
    print "str> "
```

```
    STDOUT.flush
```

```
    str = gets break if not str
```

```
str.chop!  
print "pat> "  
STDOUT.flush  
re = gets  
break if not re  
re.chop!  
str.gsub! re, "#{st}\\&#{en}"  
print str, "\n"  
  
end  
  
print "\n"
```

El programa necesita dos entradas, una con la cadena y otra con la expresión regular. La cadena se comprueba contra la expresión regular, a continuación muestra todas las partes de la cadena que coinciden con el patrón en vídeo inverso. No nos preocupemos de los detalles; analizaremos el código posteriormente.

```
str> foobar  
pat> ^fo+  
foobar  
~~~
```

Lo resaltado es lo que aparecerá en vídeo inverso como resultado de la ejecución del programa. La cadena '~~~' es en beneficio de aquellos que usen visualizadores en modo texto.

Probemos algunos ejemplos más.

```
str> abc012dbcd555  
pat> \d  
abc012dbcd555
```

~~~    ~~~

Sorprendentemente y como indica la tabla al principio de este capítulo: `\d` no tiene nada que ver el carácter `d`, sino que realiza la coincidencia con un dígito.

¿Qué pasa si hay más de una forma de realizar la coincidencia con el patrón?.

```
str> foozbooz
```

```
pat> f.*z
```

```
foozbooz
```

~~~~~

se obtiene *foozbooz* en vez de *fooz* porque las expresiones regulares tratan de obtener la coincidencia más larga posible. A continuación se muestra un patrón para aislar la hora de un campo limitada por dos puntos.

```
str> WedFeb 7 08:58:04 JST 2001
```

```
pat> [0-9]+:[0-9]+(:[0-9]+)?
```

```
WedFeb 7 08:58:04 JST 2001
```

~~~~~

`=~` es el operador de coincidencia con expresiones regulares; devuelve la posición en la cadena donde se ha producido una coincidencia o **nil** si no la hay.

```
ruby> "abcdef" =~ /d/
```

```
3
```

```
ruby> "aaaaaa" =~ /d/
```

```
nil
```

## Arrays

Se pueden crear un array listando elementos entre corchetes ([ ]) y separándolos por comas. Los arrays en Ruby pueden almacenar objetos de diferentes tipos.

```
ruby> ary = [1, 2, "3"]
```

```
[1, 2, "3"]
```

Los arrays se pueden concatenar y repetir, igual que las cadenas.

```
ruby> ary + ["foo", "bar"]
```

```
[1, 2, "3", "foo", "bar"]
```

```
ruby> ary * 2
```

```
[1, 2, "3", 1, 2, "3"]
```

Se pueden utilizar índices numéricos para acceder a cualquier parte del array.

```
ruby> ary[0]
```

```
1
```

```
ruby> ary[0,2]
```

```
[1, 2]
```

```
ruby> ary[-2]
```

```
2
```

```
ruby> ary[-2,2]
```

```
[2, "3"]
```

```
ruby> ary[-2..-1]
```

```
[2, "3"]
```

(Los índices negativos indican que se empieza a contar desde el final del array, en vez del principio). Los arrays se pueden convertir a y obtener de cadenas utilizando **join** y **split** respectivamente:

```
ruby> str = ary.join(':')
```

"1:2:3"

```
ruby> str.split(':')
```

```
["1", "2", "3"]
```

## Hashes

Un array asociativo contiene elementos que se pueden acceder, no a través de índices numéricos secuenciales, sino a través de claves que pueden tener cualquier tipo de valor. Estos arrays se conocen a veces como hash o diccionario; en el mundo Ruby se prefiere el término hash. Los hash se pueden crear mediante pares de elementos dentro de llaves ({}). Se usa la clave para encontrar algo en un hash de la misma forma que se utiliza el índice para encontrar algo en un array.

```
ruby> h = {1 => 2, "2" => "4"}
```

```
{"2"=>"4", 1=>2}
```

```
ruby> h[1]
```

```
2
```

```
ruby> h["2"]
```

```
"4"
```

```
ruby> h[5]
```

```
nil
```

```
ruby> h[5] = 10 # añadimos un valor
```

```
10
```

```
ruby> h
```

```
{5=>10, "2"=>"4", 1=>2}
```

```
ruby> h[1]=nil # borramos un valor
```

```
nil
```

```
ruby> h[1]
```

nil

ruby> h

{5=>10, "2"=>"4", 1=>nil}

## *Retomando Los Ejemplos Sencillos*

Vamos ahora a desmontar el código de nuestros anteriores programas ejemplo. Para que sirva de referencia vamos a numerar las líneas de todos los guiones.

### Factoriales

El siguiente guión aparece en el capítulo Ejemplos sencillos.

01 def fact(n)

02 if n == 0

03 1

04 else

05 n \* fact(n-1) 06 end 07 end 08 print fact(ARGV[0].to\_i), "\n"

Debido a que es la primera explicación de un código, vamos a ir línea por línea.

01 def fact(n)

En la primera línea, **def** es la sentencia que define una función (o con mayor precisión, un método; trataremos con más detalle los métodos en un capítulo posterior). Aquí se indica que la función **fact** toma un único argumento, que se llama **n**.

02 if n == 0

Con **if** comprobamos una condición. Si la condición es cierta, se evalúa la siguiente línea; si no independientemente de lo que sigue se evalúa el **else**

03 1

Si la condición es cierta el valor del if es 1.

#### 04 else

Si la condición no es cierta, se evalúa el código desde esta línea hasta el end.

#### 05 **n \* fact(n - 1)**

Si no se satisface la condición el valor de if es el resultado de multiplicar fact(n-1) por n.

#### 06 end

El primer end cierra la sentencia if

#### 07 end

El segundo end cierra la sentencia def.

#### 08 **print fact(ARGV[0].to\_i), "\n"**

Llama a la función **fact()** con el argumento especificado en la línea de comandos, e imprime el resultado.

**ARGV** es un array que contiene los argumentos de la línea de comandos. Los miembros de **ARGV** son cadenas por lo que hay que convertirlos a números enteros con **to\_i**. Ruby no convierte automáticamente las cadenas a números como hace Perl.

## Estructuras de control

### Case

Se utiliza la sentencia case para comprobar una secuencia de condiciones. Superficialmente se parece al switch de C y Java pero es considerablemente más potente como veremos.

```
ruby> i=8
8
ruby> case i
ruby| when 1, 2..5
ruby|   print "1..5\n"
ruby| when 6..10
ruby|   print "6..10\n"
ruby| end
6..10
nil
```

`2..5` es una expresión que representa un rango entre 2 y 5 inclusive. La siguiente expresión verifica si el valor `i` cae dentro del rango:

```
(2..5) === i
```

La sentencia case utiliza internamente el operador `===` para verificar las distintas condiciones. Dentro de la naturaleza orientada a objetos de Ruby, `===` lo interpreta el objeto que aparece en la condición when. Por ejemplo, el código que sigue comprueba en el primer when la igualdad de cadenas y en el segundo la coincidencia con una expresión regular.



```
ruby> case 'abcdef'
ruby| when 'aaa', 'bbb'
ruby| print "aaa o bbb\n"
ruby| when /def/
ruby| print "incluye /def/\n"
ruby| end
incluye /def/
nil
```

## while

Ruby proporciona medios adecuados para la construcción de bucles, aunque veremos en el siguiente capítulo que si se aprende a utilizar los *iteradores* a menudo hace innecesario su utilización explícita.

**while** e **if** se pueden aplicar fácilmente a sentencias individuales:

```
ruby> i = 0
0
ruby> print " Es cero.\n" if i == 0
Es cero.
nil
ruby> print "Es negativo\n" if i < 0
nil
```

```
ruby> print "#{i+=1}\n" while i < 3
```

```
1
```

```
2
```

```
3
```

```
nil
```

Algunas veces se necesita la condición de comprobación negada. Un unless es un if negado y un until es un while negado. Dejamos estas sentencias para que se experimente con ellas.

Existen cuatro formas de interrumpir el progreso de un bucle desde su interior. La primera break, como en C, sale completamente del bucle. La segunda next salta al principio de la siguiente iteración del bucle (se corresponde con la sentencia continue del C). La tercera redo reinicia la iteración en curso. A continuación se muestra un extracto de código en C que ilustra el significado de break, next, y redo:

```
while (condicion) {
```

```
label_redo:
```

```
goto label_next    /*next*/
```

```
goto label_break   /*break*/
```

```
goto label_redo    /*redo*/
```

```
;
```

```
;
```

```
label_next:
```

```
}
```

```
label_break:
```

```
;
```

La cuarta forma de salir del interior de un bucle es **return**. La evaluación de **return** provoca la salida no sólo del bucle sino también del método que contiene el bucle. Si se le pasa un argumento, lo devolverá como retorno de la llamada al método, si no el retorno será **nil**.

## For

Los programadores en C se estarán preguntando cómo se construye un bucle for. En Ruby, el bucle for es mucho más interesante de lo que cabía esperar. El siguiente bucle se ejecuta una vez por cada elemento de la colección.

```
for elem in coleccion
```

```
  ...
```

```
end
```

La colección puede ser un rango de valores (esto es lo que la mayoría de la gente espera cuando se habla de bucles for):

```
ruby> for num in (4..6)
```

```
ruby| print num, "\n"
```

```
ruby| end
```

```
4
```

```
5
```

```
6
```

```
4..6
```

Puede ser cualquier tipo de colección como por ejemplo un array:

```
ruby> for elm in [100,-9.6,"pickle"]
ruby|      print "#{elm}\\t(#{elm.type})\\n"
ruby| end
100      (Fixnum)
-9.6     (Float)
pickle   (String)
[100, -9.6, "pickle"]
```

Saliendonos un poco del tema, for es realmente otra forma de escribir each, el cual es nuestro primer ejemplo de iterador. Las siguientes dos estructuras son equivalentes:

**# Si utilizas C o Java, puedes preferir esta estructura**

**for i in coleccion**

..

**end**

**# si utilizas Smalltalk, puedes preferir esta otra**

**coleccion.each {|i|**

...

**}**

Con frecuencia se puede sustituir los bucles convencionales por iteradores y una vez acostumbrado a utilizarlos es generalmente más sencillo tratar con éstos. Por lo tanto, avancemos y aprendamos más sobre ellos.

## Iteradores

Los iteradores no son un concepto original de Ruby. Son comunes en otros lenguajes orientados a objetos. También se utilizan en Lisp aunque no se les conoce como iteradores. Sin embargo este concepto de iterador es muy poco familiar para muchas personas por lo que se explorará con detalle.

Como ya se sabe, el verbo iterar significa hacer la misma cosa muchas veces, por lo tanto un iterador es algo que hace la misma cosa muchas veces.

Al escribir código se necesitan bucles en diferentes situaciones. En C, se codifican utilizando `for` o `while`. Por ejemplo:

```
char *str;
for (str = "abcdefg"; *str != '\0'; str++) {
    /* aquí procesamos los caracteres */
}
```

La sintaxis del `for(...)` de C nos dota de una abstracción que nos ayuda en la creación de un bucle pero, la comprobación de si `*str` es la cadena nula requiere que el programador conozca los detalles de la estructura interna de una cadena. Esto hace que C se parezca a un lenguaje de bajo nivel. Los lenguajes de alto nivel se caracterizan por un soporte más flexible a la iteración. Consideremos el siguiente guión de la shell `sh`:

```
#!/bin/sh
for i in *.ch; do
    # ... aquí se haría algo con cada uno de los ficheros
done
```

Se procesarán todos los ficheros fuentes en C y sus cabeceras del directorio actual, el comando de la shell se encargaría de los detalles de coger y sustituir los nombres de los ficheros uno por uno. Pensamos que este es un método de trabajo a nivel superior que C, ¿Verdad?

Pero hay más cosas a tener en cuenta: aunque está bien que un lenguaje tenga iteradores para todos los tipos de datos definidos en él, es decepcionante tener que volver a escribir bucles de bajo nivel para los tipos de datos propios. En la POO, los usuarios definan sus propios tipos de datos a partir de otros, por lo tanto, esto puede ser un problema serio.

Luego, todos los lenguajes OO incluyen ciertas facilidades de iteración. Algunos lenguajes proporcionan clases especiales con este propósito; Ruby nos permite definir directamente iteradores.

El tipo strings de Ruby tiene algunos iteradores útiles:

```
ruby> "abc".each_byte{|c| printf"%c", c}; print "\n"
{a}{b}{c}
nil
```

**each\_byte** es un iterador sobre los caracteres de una cadena. Cada carácter se sustituye en la variable local *c*. Esto se puede traducir en algo más parecido a C ...

```
ruby> s="abc";i = 0
0
ruby> while i < s.length
ruby| printf "%c",s[i]; i+=1
ruby| end; print "\n"
{a}{b}{c}
nil
```

... sin embargo el iterador **each\_byte** es a la vez conceptualmente más simple y tiene más probabilidades de seguir funcionando correctamente incluso cuando, hipotéticamente, la clase **string** se modifique radicalmente en un futuro. Uno de los beneficios de los iteradores es que tienden a ser robustos frente a tales cambios, además, ésta es una característica del buen código en general. (Si, tengamos paciencia también hablaremos de lo que son las clases)

**each\_line** es otro iterador de **String**.

```
ruby> "a\nb\nc\n".each_line{|l| print l}
```

a

b

c

```
"a\nb\nc\n"
```

Las tareas que más esfuerzo llevan en C (encontrar los delimitadores de línea, generar subcadenas, etc.) se evitan fácilmente utilizando iteradores.

La sentencia **for** que aparece en capítulos previos itera como lo hace el iterador **each**. El iterador **each** de **String** funciona de igual forma que **each\_line**, reescribamos ahora el ejemplo anterior con un **for**:

```
ruby> for l in "a\nb\nc\n"
```

```
ruby|  print l
```

```
ruby| end
```

a

b

c

```
"a\nb\nc\n"
```

Se puede utilizar la sentencia de control **retry** junto con un bucle de iteración y se repetirá la iteración en curso desde el principio.

```
ruby> c = 0
0
ruby> for i in 0..4
ruby|   print i
ruby|   if i == 2 and c == 0
ruby|       c = 1
ruby|       print "\n"
ruby|       retry
ruby|   end
ruby| end; print "\n"
012
01234
nil
```

A veces aparece **yield** en la definición de un iterador. **yield** pasa el control al bloque de código que se pasa al iterador (esto se explorará con más detalle es el capítulo sobre los objetos procedimiento).

El siguiente ejemplo define el iterador **repeat**, que repite el bloque de código el número de veces especificado en el argumento.

```
ruby> def repeat(num)
ruby|   while num > 0
ruby|       yield
ruby|       num -= 1
ruby|   end
ruby| end
nil
ruby> repeat(3){ print "foo\n" }
foo
foo
```



**foo**

**nil**

Con `retry` se puede definir un iterador que funciona igual que `while`, aunque es demasiado lento para ser práctico.

```
ruby> def WHILE(cond)
```

```
ruby|   return if not cond
```

```
ruby|   yield
```

```
ruby|   retry
```

```
ruby| end
```

**nil**

```
ruby> i=0;WHILE(i<3){ print i; i+=1 }
```

**nil**

## **Métodos**

¿Qué es un método? En la programación OO no se piensa en operar sobre los datos directamente desde el exterior de un objeto; si no que los objetos tienen algún conocimiento de cómo se debe operar sobre ellos (cuando se les pide amablemente). Podríamos decir que se pasa un mensaje al objeto y este mensaje obtiene algún tipo de acción o respuesta significativa. Esto debe ocurrir sin que tengamos necesariamente algún tipo de conocimiento o nos importe como realiza el objeto, interiormente, el trabajo. Las tareas que podemos pedir que un objeto realice (o lo que es lo mismo, los mensajes que comprende) son los *métodos*.

En Ruby, se llama a un método con la notación punto (como en C++ o Java). El objeto con el que nos comunicamos se nombra a la izquierda del punto.

```
ruby> "abcdef".length -> 6
```

Intuitivamente, a este objeto cadena se le está pidiendo que diga la longitud que tiene. Técnicamente, se está llamando al método **length** del objeto **"abcdef"**.

Otros objetos pueden hacer una interpretación un poco diferente de **length**. La decisión sobre cómo responder a un mensaje se hace al vuelo, durante la ejecución del programa, y la acción a tomar puede cambiar dependiendo de la variable a que se haga referencia.

```
ruby> foo = "abc"
```

```
"abc"
```

```
ruby> foo.length
```

```
3
```

```
ruby> foo = ["abcde", "fghij"] ["abcde", "fghij"]
```

```
ruby> foo.length
```

```
2
```

Lo que indicamos con **length** puede variar dependiendo del objeto con el que nos comunicamos. En el primer ejemplo le pedimos a **foo** su longitud, como referencia a una cadena simple, sólo hay una respuesta posible. En el segundo ejemplo, **foo** referencia a un array, podríamos pensar que su longitud es 2, 5 ó 10; pero la respuesta más plausible es 2 (los otros tipos de longitud se podrían obtener si se desea)

```
ruby> foo[0].length
```

```
5
```

```
ruby> foo[0].length + foo[1].length
```

```
10
```

Lo que hay que tener en cuenta es que, el array *conoce lo que significa ser un array*. En Ruby, las piezas de datos llevan consigo ese conocimiento por lo que las solicitudes que se les hace se pueden satisfacer en las diferentes formas adecuadas. Esto libera al

programador de la carga de memorizar una gran cantidad de nombres de funciones, ya que una cantidad relativamente pequeña de nombre de métodos, que corresponden a conceptos que sabemos como expresar en lenguaje natural, se pueden aplicar a diferentes tipos de datos siendo el resultado el que se espera. Esta característica de los lenguajes OO (que, IMHO , Java ha hecho un pobre trabajo en explotar), se conoce como *polimorfismo* cuando un objeto recibe un mensaje que no conoce, “salta” un error:

```
ruby> foo = 5
```

```
5
```

```
ruby> foo.length
```

```
ERR: (eval):1: undefined method 'length' for 5:Fixnum
```

Por lo tanto hay que conocer qué métodos son aceptable para un objeto, aunque no se necesita saber cómo son procesados.

Si se pasan argumentos a un método, éstos van normalmente entre paréntesis.

```
objeto.metodo(arg1, arg2)
```

pero se pueden omitir, si su ausencia no provoca ambigüedad

```
objeto.metodo arg1, arg2
```

En Ruby existe una variable especial **self** que referencia al objeto que llama a un método. Ocurre con tanta frecuencia que por conveniencia se omite en las llamadas de un método dentro de un objeto a sus propios métodos:

```
self.nombre_de_metodo(args ...)
```

es igual que:

```
nombre_de_metodo(args ...)
```

Lo que conocemos tradicionalmente como *llamadas a funciones* es esta forma abreviada de llamar a un método a través de **self**. Esto hace que a Ruby se le conozca como un lenguaje orientado a objetos puro.

Aún así, los métodos funcionales se comportan de una forma muy parecida a las funciones de otros lenguajes de programación en beneficio de aquellos que no asimilen que las llamadas a *funciones* son realmente llamadas a métodos en Ruby. Se puede hablar de funciones como si no fuesen realmente métodos de objetos, si queremos.

## Clases

El mundo real está lleno de objetos que podemos clasificar. Por ejemplo, un niño muy pequeño es probable que diga “guau guau” cuando vea un perro, independientemente de su raza; naturalmente vemos el mundo en base a estas categorías.

En terminología OO, una categoría de objetos, como “perro”, se denomina clase y cualquier objeto determinado que pertenece a una clase se conoce como instancia de esa clase.

Generalmente, en Ruby y en cualquier otro lenguaje OO, se define primero las características de una clase, luego se crean las instancias. Para mostrar el proceso, definamos primero una clase muy simple Perro.

```
ruby> class Perro
ruby|   def ladra
ruby|       print "guau guau\n"
ruby|   end
ruby| end
nil
```

En Ruby, la definición de una clase es la región de código que se encuentra entre las palabras reservadas `class` y `end`. Dentro de esta área, `def` inicia la definición de un método, que como se dijo en el capítulo anterior, corresponde con algún comportamiento específico de los objetos de esa clase. Ahora que tenemos definida la clase `Perro`, vamos a utilizarla:

```
ruby> rufi = Perro.new
```

```
#<Perro: 0x401c444c>
```

Hemos creado una instancia nueva de la clase `Perro` y le hemos llamado `rufi`. El método `new` de cualquier clase, crea una nueva instancia. Dado que `rufi` es un `Perro`, según la definición de la clase, tiene las propiedades que se decidió que un `Perro` debía tener. Dado que la idea de Perrunidad es muy simple, sólo hay una cosa que puede hacer `rufi`

```
ruby> rufi.ladra
```

```
guau guau
```

```
nil
```

La creación de una instancia de una clase se conoce, a veces, como instanciación. Es necesario tener un perro antes de experimentar el placer de su conversación; no se puede pedir simplemente a la clase `Perro` que ladre para nosotros:

```
ruby> Perro.ladra
```

```
ERR: (eval):1: undefined method 'ladra' for Perro:Class
```

Tiene el mismo sentido que intentar comer el concepto de un sándwich. Por otro lado, si queremos oír el sonido de un perro sin estar emocionalmente atados, podemos crear (instanciar) un perro efímero, temporal y obtener un pequeño sonido antes de que desaparezca.

```
ruby> (Perro.new).ladra # o también, Perro.new.ladra
```

```
guau guau
```

```
nil
```

## Herencia

La clasificación de los objetos en nuestra vida diaria es evidentemente jerárquica. Sabemos que todos los gatos son mamíferos y que todos los mamíferos son animales. Las clases inferiores heredan características de las clases superiores a las que pertenecen. Si todos los mamíferos respiran, entonces los gatos respiran.

Este concepto se puede expresar en Ruby:

```
ruby> class Mamifero  
ruby| def respira  
ruby| print "inhalar y exhalar\n"  
ruby| end  
ruby| end  
nil  
  
ruby> class Gato<Mamifero  
ruby| def maulla  
ruby| print "miau \n"  
ruby| end  
ruby| end  
nil
```

Aunque no se dice cómo respira un Gato, todo gato heredará ese comportamiento de Mamifero dado que se ha definido Gato como una subclase de Mamifero. En terminología OO, la clase inferior es una subclase de la clase superior que es una superclase.

Por lo tanto, desde el punto de vista del programador, los gatos obtienen gratuitamente la capacidad de respirar; a continuación se añade el método maulla, así nuestro gato puede respirar y maullar.

```
ruby> tama = Gato.new
```

```
#<Gato:0x401c41b8>
```

```
ruby> tama.respira
```

```
inhalar y exhalar
```

```
nil
```

```
ruby> tama.maulla
```

```
miau
```

Existen situaciones donde ciertas propiedades de las superclases no deben heredarse por una determinada subclase.

Aunque en general los pájaros vuelan, los pingüinos es una subclase de los pájaros que no vuelan.

```
ruby> class Pajaro
```

```
ruby| def aseo
```

```
ruby| print "me estoy limpiando las plumas."
```

```
ruby| end
```

```
ruby| def vuela
```

```
ruby| print "estoy volando."
```

```
ruby| end
```

```
ruby| end
```

```
nil
```

```
ruby> class Pinguino<Pajaro
```

```
ruby| def vuela
```

```
ruby| fail "Lo siento. yo sólo nado."
```

```
ruby| end
```

```
ruby| end
```

```
nil
```

En vez de definir exhaustivamente todas las características de cada nueva clase, lo que se necesita es añadir o redefinir las diferencias entre cada subclase y superclase. Esta utilización de la herencia se conoce como programación *diferencial*. Y es uno de los beneficios de la programación orientada a objetos.

### ***Redefinición de métodos***

En una subclase se puede modificar el comportamiento de las instancias redefiniendo los métodos de la superclase.

```
ruby> class Humano
```

```
ruby| def identidad
```

```
ruby| print "soy una persona.\n"
```

```
ruby| end
```

```
ruby| def tarifa_tren(edad)
```

```
ruby| if edad < 12
```

```
ruby| print "tarifa reducida.\n"
```

```
ruby| else
```

```
ruby| print "tarifa normal. \n"
```

```
ruby| end
```

```
ruby| end
```

```
ruby| end
```



**nil**

**ruby> Humano.new.identidad**

**soy una persona.**

**nil**

**ruby> class Estudiante<Humano**

**ruby| def identidad**

**ruby| print "soy un estudiante.\n"**

**ruby| end**

**ruby| end**

**nil**

**ruby> Estudiante.new.identidad**

**soy un estudiante.**

**nil**

Supongamos que en vez de reemplazar el método identidad lo que queremos es mejorarlo. Para ello podemos utilizar

**super**

**ruby> class Estudiante2<Humano**

**ruby| def identidad**

**ruby| super**

**ruby| print "también soy un estudiante.\n"**

**ruby| end**

**ruby| end**

**nil**

**ruby> Estudiante2.new.identidad**

**soy una persona.**

**también soy un estudiante.**

**nil**

super nos permite pasar argumentos al método original. Se dice que hay dos tipos de personas ...

```
ruby> class Deshonesta<Humano
```

```
ruby| def tarifa_tren(edad)
```

```
ruby| super(11)      #quiero una tarifa barata
```

```
ruby| end
```

```
ruby| end
```

**nil**

```
ruby> Deshonesta.new.tarifa_tren(25)
```

**tarifa reducida.**

**nil**

```
ruby> class Honesta<Humano
```

```
ruby| def tarifa_tren(edad)
```

```
ruby| super(edad) #pasa el argumento entregado
```

```
ruby| end
```

```
ruby| end
```

**nil**

```
ruby> Honesta.new.tarifa_tren(25)
```

**tarifa normal.**

**nil**

## Control de accesos

Se ha dicho anteriormente que Ruby no tiene funciones, sólo métodos. Sin embargo existe más de una clase de métodos. En este capítulo vamos a presentar el control de accesos.

Vamos a considerar lo que pasa cuando se define un método en el “nivel superior”, no dentro de una clase. Se puede pensar que dicho método es análogo a una función de un lenguaje más tradicional como C.

```
ruby> def square(n)
```

```
ruby| n * n
```

```
ruby| end
```

```
nil
```

```
ruby> square(5)
```

```
25
```

Nuestro nuevo método parece que no pertenece a ninguna clase, pero de hecho Ruby se lo asigna a la clase Object, que es la superclase de cualquier otra clase. Como resultado de esto cualquier objeto es capaz de utilizar este método. Esto es cierto, pero existe un pequeño pero; es un método privado a cada clase. A continuación hablaremos más de lo que esto significa, pero una de sus consecuencias es que sólo se puede llamar de la siguiente forma

```
ruby> class Foo
```

```
ruby| def fourth_power_of (x)
```

```
ruby| square(x) * square(x)
```

```
ruby| end
```

```
ruby| end nil
```

```
ruby> Foo.new.fourth_power_of 10
```

**10000**

No se nos permite aplicar explícitamente el método a un objeto:

**"fish".square(5)**

**ERR: (eval):1: private method 'square' called for "fish":String**

Esto preserva con inteligencia la naturaleza puramente OO de Ruby (las funciones siguen siendo métodos de objetos, donde el receptor implícito es self), a la vez que proporciona funciones que se pueden escribir de igual forma que en lenguajes tradicionales.

Una disciplina mental común en la programación OO, que ya se señaló en un capítulo anterior, tiene que ver con la separación de la especificación y la implementación o qué tareas se supone que un objeto realiza y cómo realmente se consiguen. El trabajo interno de un objeto debe mantenerse, por lo general, oculto a sus usuarios; sólo se tiene que preocupar de lo que entra y lo que sale y confiar en que el objeto sabe lo que está realizando internamente.

Así, es generalmente útil que las clases posean métodos que el mundo exterior no ve, pero que se utilizan internamente (y que pueden ser mejorados por el programador cuando desee, sin modificar la forma en que los usuarios ven los objetos de esa clase). En el trivial ejemplo que sigue, piénsese que engine es el motor interno de la clase.

**ruby> class Test**

**ruby| def times\_two(a)**

**ruby| print a," dos veces es ",engine(a),"\\n"**

**ruby| end**

**ruby| def engine(b)**

**ruby| b\*2**

**ruby| end**

```
ruby| private:engine # esto oculta engine a los usuarios
```

```
ruby| end
```

```
Test
```

```
ruby> test = Test.new
```

```
# <Test:0x401c4230>
```

```
ruby> test.engine(6)
```

```
ERR: (eval):1: private method 'engine' called for #<Test:v0x401c4230>
```

```
ruby> test.times_two(6)
```

```
6 dos veces es 12
```

```
nil
```

Se podría esperar que **test.engine(6)** devolviese 12, pero por el contrario se nos comunica que engine es inaccesible cuando actuamos como usuario del objeto **Test**. Sólo otros métodos de **Test**, como **times\_two** tienen permiso para utilizar engine. Se nos obliga a pasar por el interfaz público, que es el método **times\_two**. El programador que está al cargo de la clase puede modificar **engine** (en este caso cambiando **b\*2** por **b+b** suponiendo que así mejora el rendimiento) sin afectar cómo los usuarios interactúan con el objeto **Test**. Este ejemplo, por supuesto, es demasiado simple para ser útil; los beneficios de control de accesos se manifiestan cuando se comienzan a crear clases más complicadas e interesantes.

## Módulos

Los módulos en Ruby son similares a las clases, excepto en:

- Un módulo no puede tener instancias
- Un módulo no puede tener subclases
- Un módulo se define con **module ... end**

Ciertamente ... la clase Module de un módulo es la superclase de la clase Class de una clase. ¿Se pillan estas? ¿No? Sigamos.

Existen dos usos típicos de los módulos. Uno es agrupar métodos y constantes relacionadas en un repositorio central. El módulo **Math** de Ruby hace esta función:

```
ruby> Math.sqrt(2)
```

```
1.414213562
```

```
ruby> Math::PI
```

```
3.141592654
```

El operador `::` indica al intérprete de Ruby qué módulo debe consultar para obtener el valor de la constante (es concebible, que algún otro módulo a parte de Math intérprete PI de otra forma).

Si queremos referenciar a los métodos o constantes de un módulo, directamente, sin utilizar `::`, podemos incluir ese módulo con `include`:

```
ruby> include Math
```

```
ruby> sqrt(2) 1.414213562
```

```
ruby> PI 3.141592654
```

El otro uso de los módulos se denomina mixin. Algunos lenguajes OO, incluidos el C++, permiten herencia múltiple, es decir, una clase puede heredar de más de una superclase. Un ejemplo de herencia múltiple en el mundo real es un despertador, se podría pensar que un despertador es una clase de *reloj* y que también pertenece a la clase de objetos que podríamos llamar *zumbadores*.

Ruby, con toda la intención del mundo, no implementa herencia múltiple real, aunque la técnica de los mixins es una buena alternativa. Recuérdese que los módulos no se pueden instanciar ni se pueden crear subclases de ellos; pero si se incluye un módulo en la definición de una clase sus métodos quedan añadidos a ella, es decir se asocian (mixin1 ) a la clase.

Se puede pensar que los mixins son una forma de pedir qué propiedades concretas se desean. Por ejemplo, si una clase tiene un método **each** funcional, asociarla con el módulo **Enumerable** de la biblioteca estándar nos proporciona gratuitamente los métodos **sort** y **find**.

Esta utilización de los módulos proporciona la funcionalidad básica de la herencia múltiple permitiéndonos representar las relaciones de la clase en una simple estructura en árbol que simplifica considerablemente la implementación del lenguaje (Los diseñadores de Java hicieron una elección parecida).

## Variables

Ruby tiene tres clases de variables, una clase de constante y exactamente dos pseudo-variables. Las variables y las constantes no tienen tipo. Aunque las variables sin tipo tienen sus inconvenientes, presentan más ventajas y se adaptan mejor a la filosofía *rápido y sencillo* de Ruby

En la mayoría de los lenguajes hay que declarar las variables para especificar su tipo, si se pueden modificar (e.g. si son constantes) e indicar su ámbito, ya que no es ningún problema el tipo y como vamos a ver, el resto se obtiene a partir del nombre, en Ruby no se necesita declarar las variables.

El primer carácter de un identificador lo cataloga de un plumazo:

| Símbolo   | Tipo de Variable   |
|-----------|--------------------|
| \$        | Variable Global    |
| @         | Variable Instancia |
| [a-z] ó _ | Variable Local     |
| [A-Z]     | Constante          |

Tabla 6 .- Variables

## Variables Globales

Una variable global tiene un nombre que comienza con \$. Se puede utilizar en cualquier parte de un programa. Antes de inicializarse, una variable global tiene el valor especial nil.

```
ruby> $foo
```

```
nil
```

```
ruby> $foo = 5
```

```
5
```

```
ruby> $foo
```

```
5
```



Las variables globales deben utilizarse con parquedad. Son peligrosas porque se pueden modificar desde cualquier lugar. Una sobreutilización de variables globales puede dificultar la localización de errores; también indica que no se ha pensado detenidamente el diseño del programa. Siempre que se encuentre la necesidad de utilizar una variable global, hay que darle un nombre descriptivo para que no se pueda utilizar inadvertidamente para otra cosa (Llamarle **\$foo** como se ha hecho en el ejemplo es probablemente una mala idea).

Una característica notable de las variables globales es que se pueden trazar; se puede definir un procedimiento que se llame cada vez que se modifique el valor de la variable.

```
ruby> trace_var:$x, proc{print "$x es ahora ", $x, "\n"}
```

```
nil
```

```
ruby> $x = 5
```

```
$x es ahora 5
```

```
5
```

Cuando una variable global se la atavía para que funcione con un disparador que se llama cada vez que se modifica, se la conoce como variable activa. Son útiles, por ejemplo, para mantener un GUI actualizado.

Existe un grupo especial de variables cuyos nombres constan del símbolo del dolar (\$) seguido de un carácter. Por ejemplo, **\$\$** contiene el número de identificación del proceso del intérprete de Ruby, y es de sólo lectura.

A continuación se muestran las principales variables del sistema y su significado.

| Símbolo | Descripción                                                                            |
|---------|----------------------------------------------------------------------------------------|
| \$!     | Último mensaje de error                                                                |
| \$@     | Posición del error                                                                     |
| \$_     | Ultima cadena leída con gets                                                           |
| \$.     | Ultimo numero de linea leído por el intérprete                                         |
| \$&     | Última cadena que ha coincidido con una expresión regular                              |
| \$~     | Última cadena que ha coincidido con una expresión regular como array de subexpresiones |
| \$n     | La n-ésima subexpresion regular de la última coincidencia (igual que \$~[n])           |
| \$=     | flag para tratar igual las mayusculas y minusculas                                     |
| \$/     | separador de registros de entrada                                                      |
| \$\     | separador de registros de salida                                                       |
| \$()    | El nombre del fichero del guión Ruby                                                   |
| \$*     | El comando de la línea de argumentos                                                   |
| \$\$    | El número de identificación del proceso del intérprete Ruby                            |
| \$?     | Estado de retorno del último proceso hijo ejecutado                                    |

Tabla 7 .- Variables del Sistema

De las variables anteriores `$_` y `$~`, tienen ámbito local. Sus nombres sugieren que deberían tener ámbito global, pero son más útiles de esta forma y existen razones históricas para utilizar estos identificadores.

## Variables de Instancia

Una variable de instancia tiene un nombre que comienza con `@` y su ámbito está limitado al objeto al que referencia `self`.

Dos objetos diferentes, aún cuando pertenezcan a la misma clase, pueden tener valores diferentes en sus variables de instancia. Desde el exterior del objeto, las variables de instancia, no se pueden alterar e incluso, no se pueden observar (es decir, en Ruby las variables de instancia nunca son públicas) a excepción de los métodos proporcionados explícitamente por el programador. Como con las variables globales, las variables de instancia tienen el valor `nil` antes de que se inicialicen.

Las variables de instancia en Ruby no necesitan declararse. Esto da lugar a una estructura flexible de los objetos. De hecho, cada variable de instancia se añade dinámicamente al objeto la primera vez que se la referencia

```
ruby> class InstTest
ruby| def set_foo(n)
ruby|   @foo = n
ruby| end
ruby| def set_bar(n)
ruby|   @bar = n
ruby| end
ruby| end
nil
ruby> i = InstTest.new
```

```
#<InstTest:0x401c3e0c>
```

```
ruby> i.set_foo(2)
```

```
2
```

```
ruby> i
```

```
#<InstTest:0x401c3e0c @foo=2>
```

```
ruby> i.set_bar(4)
```

```
4
```

```
ruby> i
```

```
#<InstTest:0x401c3e0c @bar=4, @foo=2>
```

Obsérvese que **i** no informa del valor de **@bar** hasta que no se haya llamado al método **set\_bar**

## Variables Locales

Una variable local tiene un nombre que empieza con una letra minúscula o con el carácter de subrayado (\_). Las variables locales no tienen, a diferencia de las variables globales y las variables de instancia, el valor nil antes de la inicialización:

```
ruby> $foo
```

```
nil
```

```
ruby> @foo
```

```
nil
```

```
ruby> foo
```

```
ERR: (eval):1: undefined local variable or method 'foo'
```

La primera asignación que se realiza sobre una variable local actúa como una declaración. Si se referencia a una variable local no inicializada, el intérprete de Ruby

piensa que se trata de una llamada a un método con ese nombre; de ahí el mensaje de error del ejemplo anterior.

Generalmente el ámbito de una variable local es uno de los siguientes:

- **proc{ ... }**
- **loop{ ... }**
- **def ... end**
- **class ... end**
- **module ... end**
- **Todo el programa (si no es aplicable ninguno de los puntos anteriores)**

En el siguiente ejemplo **defined?** es un operador que verifica si un identificador está definido. Si lo está, devuelve una descripción del mismo, en caso contrario, devuelve **nil**. Como se ve el ámbito de **bar** es local al bucle, cuando se sale del bucle, **bar** está sin definir.

```
ruby> foo =44; print foo, "\n"; defined? foo
```

```
44
```

```
"local-variable"
```

```
ruby> loop{bar = 45;print bar, "\n"; break}; defined? var
```

```
45
```

```
nil
```

Los objetos procedimiento que residen en el mismo ámbito comparten las variables locales que pertenecen a ese ámbito. En el siguiente ejemplo, la variable **bar** es compartida por **main** y los objetos procedimiento **p1** y **p2**:

```
ruby> bar=0 0
```

```
ruby> p1 = proc{|n| bar = n} #
```

```
ruby> p2 = proc{bar} #
```

```
ruby> p1.call(5)
```

```
5
```

```
ruby> bar
```

```
5
```

```
ruby> p2.call
```

```
5
```

Obsérvese que no se puede omitir la línea `bar=0` inicial; esta asignación es la que garantiza que el ámbito de `bar` incluirá a `p1` y `p2`. Si no, `p1` y `p2` tendrán al final cada uno su propia variable local `bar` y la llamada a `p2` dará lugar a un error de “variable o método no definido”.

Una característica muy poderosa de los objetos procedimiento se deriva de su capacidad para recibir argumentos; las variables locales compartidas permanecen válidas incluso cuando se las pasa fuera de su ámbito original.

```
ruby> def box
```

```
ruby| contents = 15
```

```
ruby| get = proc{contents}
```

```
ruby| set = proc{|n| contents = n}
```

```
ruby| return get, set
```

```
ruby| end
```

```
nil
```

```
ruby> reader, writer = box
```

```
ruby> reader.call
```

```
15
```

```
ruby> writer.call(2)
```

2

```
ruby> reader.call
```

2

Ruby es especialmente inteligente con respecto al ámbito. En el ejemplo, es evidente que la variable `contents` está compartida por `reader` y `writer`. Ahora bien, es posible definir varios pares `reader-writer` que utilicen `box` cada uno de los cuales comparten su propia variable `contents` sin interferir uno con otro.

```
ruby> reader_1, writer_1 = box
```

```
ruby> reader_2, writer_2 = box
```

```
ruby> writer_1.call(99)
```

99

```
ruby> reader_1.call
```

99

```
ruby> reader_2.call
```

15

## **Constantes**

Una constante tiene un nombre que comienza con una letra mayúscula. Se le debe asignar valor sólo una vez. En la implementación actual de Ruby, reasignar un valor a una constante genera un aviso y no un error (la versión no ANSI de `eval.rb` no informa de este aviso):

```
ruby> fluid = 30
```

30

```
ruby> fluid = 31
```

31

```
ruby> Solid = 32
```

32

```
ruby> Solid = 33
```

```
(eval):1: warning: already initialized constant Solid
```

```
33
```

Las constantes se pueden definir en una clase, pero a diferencia de las variables de instancia, son accesibles desde el exterior de la misma.

```
ruby> class ConstClass
```

```
  ruby| C1=101
```

```
  ruby| C2=102
```

```
  ruby| C3=103
```

```
  ruby| def show
```

```
  ruby| print C1, " ", C2, " ", C3, "\n"
```

```
  ruby| end
```

```
  ruby| end
```

```
nil
```

```
ruby> C1
```

```
ERR: (eval):1: uninitialized constant C1
```

```
ruby> ConstClass::C1
```

```
101
```

```
ruby> ConstClass.new.show
```

```
101 102 103
```

```
nil
```

Las constantes también se pueden definir en un módulo.

```
ruby> module ConstModule
```

```
  ruby| C1=101
```

```
  ruby| C2=102
```

```
  ruby| C3=103
```



```
ruby| def showConstants
```

```
ruby| print C1," ",C2," ",C3,"\n"
```

```
ruby| end
```

```
ruby| end
```

```
nil
```

```
ruby> C1
```

```
ERR: (eval):1: uninitialized constant C1
```

```
ruby> include ConstModule
```

```
Object
```

```
ruby> C1
```

```
101
```

```
ruby> showConstants
```

```
101 102 103
```

```
nil
```

```
ruby> C1=99 # realmente una idea no muy buena
```

```
99
```

```
ruby> C1
```

```
99
```

```
ruby> ConstModule::C1 # La constante del módulo queda sin tocar ...
```

```
101
```

```
ruby> ConstModule::C1=99
```

```
ERR: (eval):1: compile error
```

```
(eval):1: parse error ConstModule::C1=99
```

^

```
ruby> ConstModule::C1 #... independientemente de lo que hayamos jugado con ella
```

## Fuente jj

```
//***** GRAMÁTICAS *****/
void start():{}{
    program()
}

//***** PROGRAMA *****/
void program():{}{
    (
        LOOKAHEAD(5) stmt()
    | LOOKAHEAD(5) func()
    | LOOKAHEAD(5) estructuras()
    | LOOKAHEAD(5) clases()
    | opr(<S_SALTO>
    )*
}

//***** EXPRESIONES GENERALES *****/
void opr():{}{
    try {
        LOOKAHEAD(13)varname() op_arit() varname()
    | LOOKAHEAD(3)valores() op_arit() valores()
    | LOOKAHEAD(13)varname() op_arit() valores()
    | LOOKAHEAD(2)valores() op_arit() varname()
    | LOOKAHEAD(10)varname() op_asigna() varname()
    | LOOKAHEAD(10)varname() op_asigna() valores()
    | LOOKAHEAD(13)varname() op_logicos() varname()
    | LOOKAHEAD(3)valores() op_logicos() valores()
    | LOOKAHEAD(13)varname() op_logicos() valores()
    | LOOKAHEAD(2)valores() op_logicos() varname()
    | LOOKAHEAD(13)varname() op_compara() varname()
    | LOOKAHEAD(3)valores() op_compara() valores()
    | LOOKAHEAD(13)varname() op_compara() valores()
    | LOOKAHEAD(2)valores() op_compara() varname()
    | LOOKAHEAD(13)varname() op_rango() varname()
    | LOOKAHEAD(3)valores() op_rango() valores()
    | LOOKAHEAD(13)varname() op_rango() valores()
    | LOOKAHEAD(3)valores() op_rango() varname()
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind != MCVillCompilerTokenManager.EOF);
    }
}
```

```

//***** STMT *****
void stmt():{}{
    LOOKAHEAD(5) declara()
    | LOOKAHEAD(5) imprimir()
    | LOOKAHEAD(5) accArregos()
    | LOOKAHEAD(5) parsear()
    | llamadaMetodos()
}

//***** DECLARACIONES *****
void declara(): {
    Token id;
    String type = null;
    int line = 0;
}
{
    try {
        ( id = varname()
        <OAS_ASIGNA>
        (
            LOOKAHEAD(5) rangos() <S_SALTO>
            | LOOKAHEAD(5) arreglos() <S_SALTO>
            | LOOKAHEAD(5) lectura() <S_SALTO>
            | LOOKAHEAD(5) parsear()
            | valor() <S_SALTO> { for( int i = 0; i < listValues.size(); i++ )
                                {
                                    for (int j = 0; j < listValues.size(); j++ )
                                    {
                                        type = sm.compareTypes( listValues.get( i ),
listValues.get( j ) );
                                    }
                                }
                                sm.addVariable( id, type );
                            }
                        )
                    ) {listValues.clear();}
                }
            catch (ParseException e) {
                iom.errorSyntax(e);
                Token t;
                do {
                    t = getNextToken();
                } while ( t.kind != MCVillCompilerTokenManager.EOF);
            }
            catch (SemanticException se){
                iom.errorSemantic(se);
                Token t;
                do {
                    t = getNextToken();
                } while ( t.kind != MCVillCompilerTokenManager.EOF );
            }
        }
    }
}

```

```

//***** VALORES *****/
void valor()throws SemanticException:{}
{
    expr() (op_arit() expr()*)
}

//***** EXPRESIONES *****/
void expr():{
    Token id;
}
{
    try{
        (
            id = <ID_LOCAL>    {
                                sm.checkVariable( id );
                                listValues.add( id );
                                }
            | id = <VAL_ENTERO>  { listValues.add( id ); }
            | id = <VAL_CADENA>  { listValues.add( id ); }
            | id = <VAL_DECIMAL> { listValues.add( id ); }
            | id = <VAL_CADENA2> { listValues.add( id ); }
            | "(" valor() ")"
        )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
    }
    catch (SemanticException se){
        iom.errorSemantic(se);
    }
}

//***** OPERADORES *****/
void op_arit():{}{
    ("-" | "+" | "*" | "/" | "%" | "**" )
}
void op_logicos():{}{
    ("&&" | "||" | "!" )
}
void op_rango():{}{
    (".." | "...")
}
void op_compara():{}{
    ("==" | "!=" | "<" | ">" | ">=" | "<=" | "<=>" | "?" | "===")
}
void op_asigna():{}{
    ("+=" | "-=" | "*=" | "/=" | "**=" | "%=")
}
}

```

```
//***** VARIABLES *****
```

```
Token varname():{  
    Token id;  
}  
{  
    (  
        LOOKAHEAD(5) id = <ID_LOCAL>  
    | LOOKAHEAD(5) id = <ID_CONSTANT>  
    | LOOKAHEAD(5) id = <ID_GLOBAL>  
    | LOOKAHEAD(5) id = <ID_CLASS>  
    | id = <ID_INSTANCE>  
    )  
    { return id; }  
}
```

```
//***** VALORES *****
```

```
void valores():{}{  
    (  
        <VAL_ENTERO>  
    | <VAL_DECIMAL>  
    | <VAL_CADENA>  
    | <VAL_CADENA2>  
    | <ID_LOCAL>  
    | <PR_TRUE>  
    | <PR_FALSE>  
    )  
}
```

```
//***** CONTENIDO RANGOS *****
```

```
void rangos():{}{  
    try {  
        (  
            LOOKAHEAD(3)varname() ("..."|"..") varname()  
        |valores() ("..."|"..") valores()  
        )  
    }  
    catch (ParseException e) {  
        iom.errorSyntax(e);  
        Token t;  
        do {  
            t = getNextToken();  
        } while (t.kind == MCVillCompilerTokenManager.EOF);  
        declara();  
    }  
}
```

```
//***** CONTENIDO ARREGLOS *****
```

```
void arreglos():{}{
    try {
        ( <S_CORCHETE_A> valores() (<S_COMA> valores())* <S_CORCHETE_C> )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
    }
}
```

```
//***** CONTENIDO LECTURAS *****
```

```
void lectura():{}{
    try {
        (
            varname()". "<PR_GETS> (LOOKAHEAD(5)". " <PR_CHOMP> ( ". " (<PR_TO_I> |
<PR_TO_F>)))?
            | ". " (<PR_TO_I> | <PR_TO_F>)))?
        )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
    }
}
```

```
//***** PARSEO DE VARIABLES *****
```

```
void parsear():{}{
    try {
        (
            varname()". "puntosPalabras() <S_SALTO>
        )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        stmt();
    }
}
```

```

//***** PUNTOS + PALABRAS RESERVADAS *****
void puntosPalabras():{}{
try {
    LOOKAHEAD(8)<PR_CLASS>
    | LOOKAHEAD(8)<PR_TO_S>
    | LOOKAHEAD(8)<PR_TO_F>
    | LOOKAHEAD(8)<PR_TO_I>
    | LOOKAHEAD(8)<PR_NEW>
    | LOOKAHEAD(8)<PR_EACH> <PR_DO>
    | (LOOKAHEAD(3)<PR_EMPTY> | <PR_INCLUDE> | <PR_UPCASE>)(LOOKAHEAD(2)("&?"|"!"))?
}
catch (ParseException e) {
    iom.errorSyntax(e);
    Token t;
    do {
        t = getNextToken();
    } while (t.kind == MCVillCompilerTokenManager.EOF);
    parsear();
}
}

//***** IMPRESIONES -PUTS -PRINT -P *****
void imprimir():{}{
try {
    ( LOOKAHEAD(4)<PR_PUTS>|LOOKAHEAD(4)<PR_PRINT>|LOOKAHEAD(4)<PR_PRINTF>|<PR_P> )
    (
        LOOKAHEAD(2) valores() <S_SALTO>
        | LOOKAHEAD(2) varname() <S_SALTO>
        | parsear()
    )
}
catch (ParseException e) {
    iom.errorSyntax(e);
    Token t;
    do {
        t = getNextToken();
    } while (t.kind == MCVillCompilerTokenManager.EOF);
    stmt();
}
}

//***** ACCESO ARREGLOS *****
void accArregos():{}{
try {
    ( varname() "[" (LOOKAHEAD(2)varname() | valores()) "]" <S_SALTO> )
}
catch (ParseException e) {
    iom.errorSyntax(e);
    Token t;
    do { t = getNextToken();
    } while (t.kind == MCVillCompilerTokenManager.EOF);
    stmt();
}
}

```

```
//***** LLAMADA A METODOS *****
```

```
void llamadaMetodos():{}{
    try {
        (
            varname() "(" (LOOKAHEAD(2)varname()
            | valores()) (<S_COMA> (LOOKAHEAD(2)varname()
            | valores()))* ")" <S_SALTO>
        )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        stmt();
    }
}
```

```
//***** FUNCIONES *****
```

```
void func():{}{
    try {
        ( metodos() )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        func();
    }
}
```

```
//***** METODOS *****
```

```
void metodos():{}{
    try {
        (
            <PR_DEF> (<PR_SELF> ".")? varname()
            (( "(")? varname() ("," (varname()| ("*)(<ID_LOCAL>) | ("**")(<ID_LOCAL>) ))*
            ("")? )? <S_SALTO>
            program()
            <PR_END> <S_SALTO>
        )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        func();
    }
}
```



```
//***** ESTRUCTURAS *****
```

```
void estructuras():{}{
    try {
        LOOKAHEAD(5) esIf()
        | LOOKAHEAD(5) esUnless()
        | LOOKAHEAD(5) esWhile()
        | LOOKAHEAD(5) esUntil()
        | esFor()
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        program();
    }
}
```

```
//***** IF *****
```

```
void esIf():{}{
    try {
        <PR_IF> varname() op_compara() (LOOKAHEAD(2)varname()|valores()) <S_SALTO>
            program()
        (esElsif())?
        <PR_END><S_SALTO>
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF & t.image != "\n");
        estructuras();
    }
}
```

```
//*****+***** ELSIF *****
```

```
void esElsif():{}{
    try {
        <PR_ELSIF> varname() op_compara() (LOOKAHEAD(2)varname()|valores()) <S_SALTO>
            program()
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
    }
}
```

```

//***** UNLESS *****
void esUnless():{}{
    try {
        <PR_UNLESS> varname() op_compara() (LOOKAHEAD(2)varname())|valores()) <S_SALTO>
            program()
        <PR_END><S_SALTO>
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        estructuras();
    }
}

//***** WHILE *****
void esWhile():{}{
    try {
        <PR_WHILE> opr() <S_SALTO>
            program()
        <PR_END><S_SALTO>
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        estructuras();
    }
}

//***** UNTIL *****
void esUntil():{}{
    try {
        (
            <PR_UNTIL> opr() <S_SALTO>
            program()
            <PR_END><S_SALTO>
        )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        estructuras();
    }
}

```

```

//***** FOR *****
void esFor():{}{
    try {
        (
            <PR_FOR> opr() <S_SALTO>
                program()
            <PR_END><S_SALTO>
        )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        estructuras();
    }
}

```

```

//***** CLASES *****
void clases():{}{
    try {
        (
            <PR_CLASS> varname() ("<" varname())? <S_SALTO>
                program()
            <PR_END><S_SALTO>
        )
    }
    catch (ParseException e) {
        iom.errorSyntax(e);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind == MCVillCompilerTokenManager.EOF);
        clases();
    }
}

```

# Semántica

## Descripción

Un error semántico se produce cuando la sintaxis del código es correcta, pero la semántica o el significado no es el que se pretendía.

Un error semántico puede hacer que el programa termine de forma anormal, con o sin un mensaje de error. Esto depende si se hace uso de las excepciones.

## Tratamiento de errores semánticos

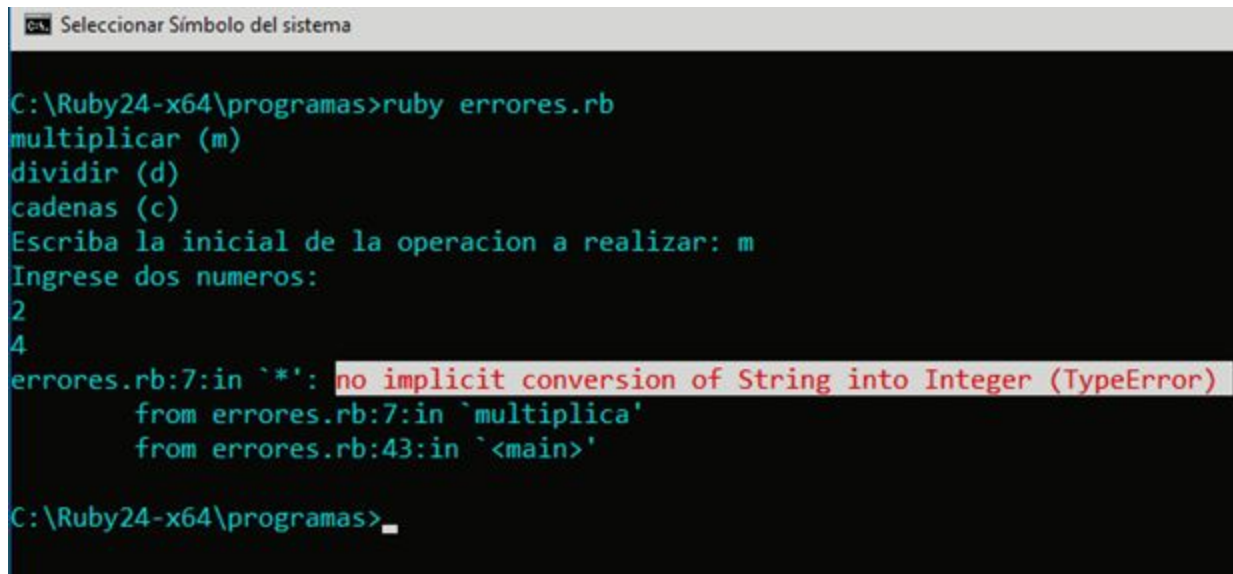
### Error de parseo

En el lenguaje Ruby las lecturas de datos por medio del teclado se realiza utilizando la función **gets**, esta función guarda la información leída como un "String".

```
40 puts "Ingrese dos numeros: "  
41 · · x=gets  
42 · · y=gets ·  
43 multiplica(x,y)
```

Figura 13 - Uso de la función gets.

## Compilación sin parseo:



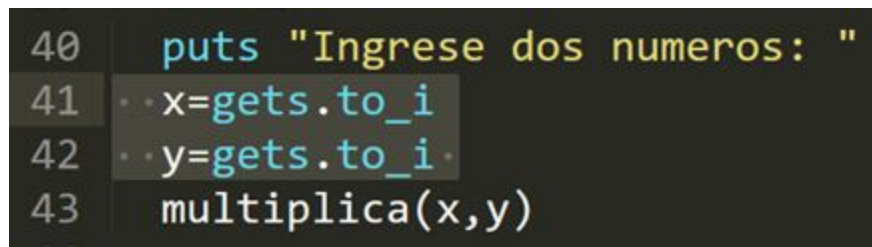
```
Seleccionar Símbolo del sistema

C:\Ruby24-x64\programas>ruby errores.rb
multiplicar (m)
dividir (d)
cadenas (c)
Escriba la inicial de la operacion a realizar: m
Ingresa dos numeros:
2
4
errores.rb:7:in `*': no implicit conversion of String into Integer (TypeError)
    from errores.rb:7:in `multiplica'
    from errores.rb:43:in `<main>'

C:\Ruby24-x64\programas>_
```

Figura 14.- Compilación sin Parseo (Error de Conversión).

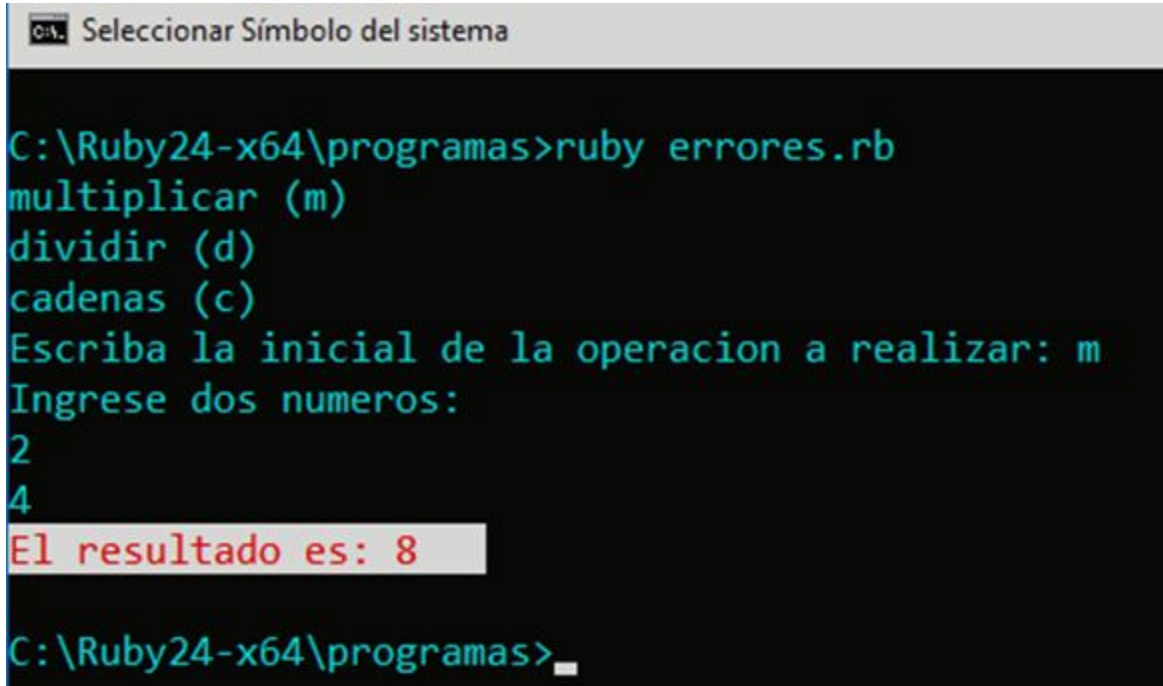
Si queremos realizar alguna operación con un tipo de dato entero o flotante es necesario realizar un parseo mediante las funciones `.to_i` (Tipo de dato entero) y `.to_f` (Tipo de dato flotante).



```
40 puts "Ingresa dos numeros: "
41 x=gets.to_i
42 y=gets.to_i
43 multiplica(x,y)
```

Figura 15.- Uso de la función `.to_i`.

## Compilación con parseo (Usando la función `.to_i`) :



```
Selecciónar Símbolo del sistema

C:\Ruby24-x64\programas>ruby errores.rb
multiplicar (m)
dividir (d)
cadenas (c)
Escriba la inicial de la operacion a realizar: m
Ingrese dos numeros:
2
4
El resultado es: 8

C:\Ruby24-x64\programas>_
```

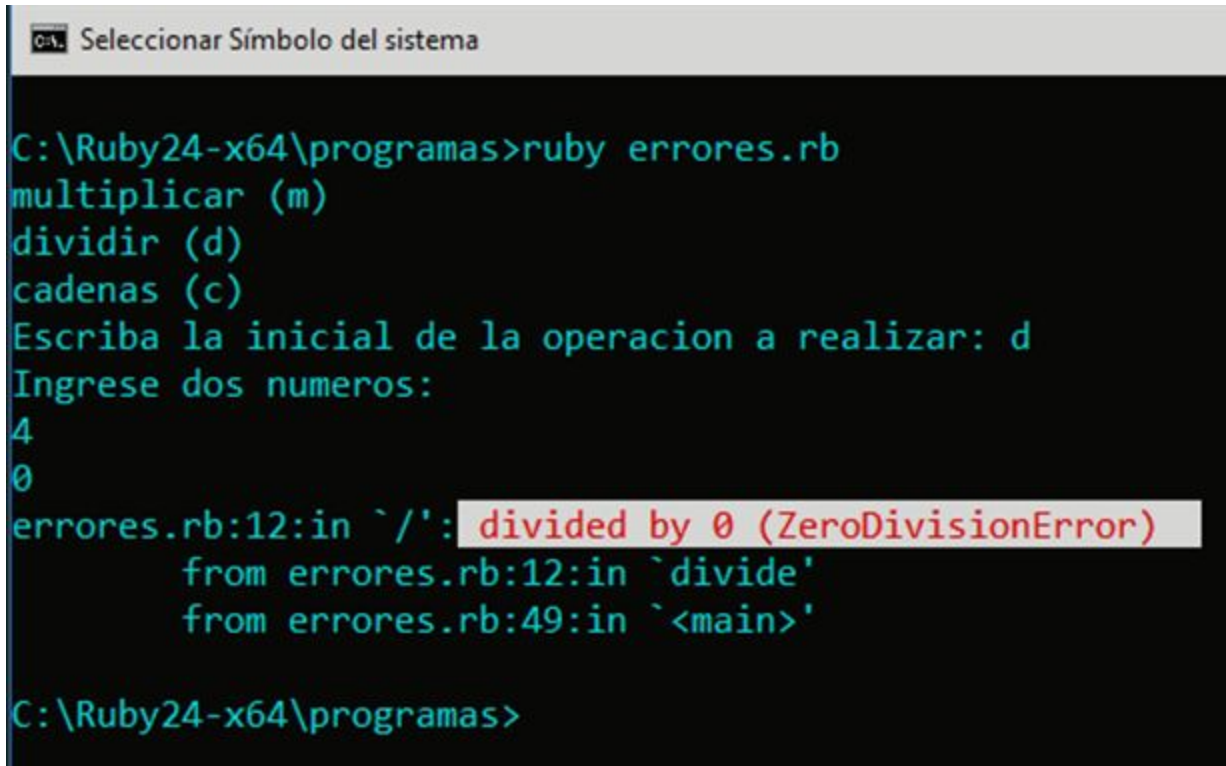
Figura 16 - Compilación con Parseo.

## División entre Cero

En aritmética y álgebra es considerada una indefinición que puede originar paradojas matemáticas.

En los números naturales, enteros y reales, la división entre cero no posee un valor definido, debido a que para todo número  $n$ , el producto  $n \cdot 0 = 0$ , por lo que el 0 no tiene inverso multiplicativo.

## Compilación división entre cero:



```
O:\ Seleccionar Símbolo del sistema

C:\Ruby24-x64\programas>ruby errores.rb
multiplicar (m)
dividir (d)
cadenas (c)
Escriba la inicial de la operacion a realizar: d
Ingrese dos numeros:
4
0
errores.rb:12:in `/' : divided by 0 (ZeroDivisionError)
    from errores.rb:12:in `divide'
    from errores.rb:49:in `<main>'

C:\Ruby24-x64\programas>
```

Figura 17.- Compilación división entre cero.

Ruby no acepta dividir un número entre cero, ya que esta división es un error semántico. **El divisor siempre debe de ser mayor a cero.**

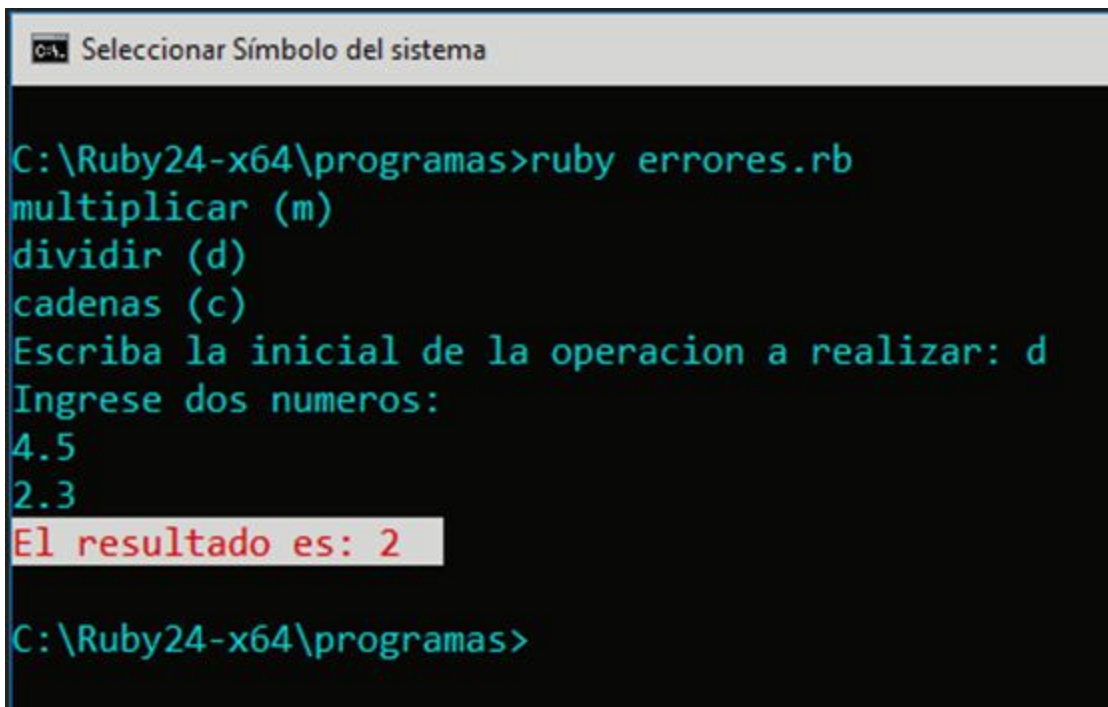
## División entre distintos tipos de datos

En el método divide tenemos las variables a y b parseadas a números enteros. Observaremos el resultado al ingresar dos números flotantes al método divide.

```
10 def divide(a,b)
11   puts "El resultado es: #{a.to_i/b.to_i}"
12 end
```

Figura 18.- Parseo a números enteros.

## Compilación división de dos valores flotantes:



```
C:\Ruby24-x64\programas>ruby errores.rb
multiplicar (m)
dividir (d)
cadenas (c)
Escriba la inicial de la operacion a realizar: d
Ingresa dos numeros:
4.5
2.3
El resultado es: 2
C:\Ruby24-x64\programas>
```

Figura 19.- Compilación división de dos números flotantes.

Ruby por defecto eliminará el punto decimal de los valores flotantes y realizará la división solo los con los números enteros.



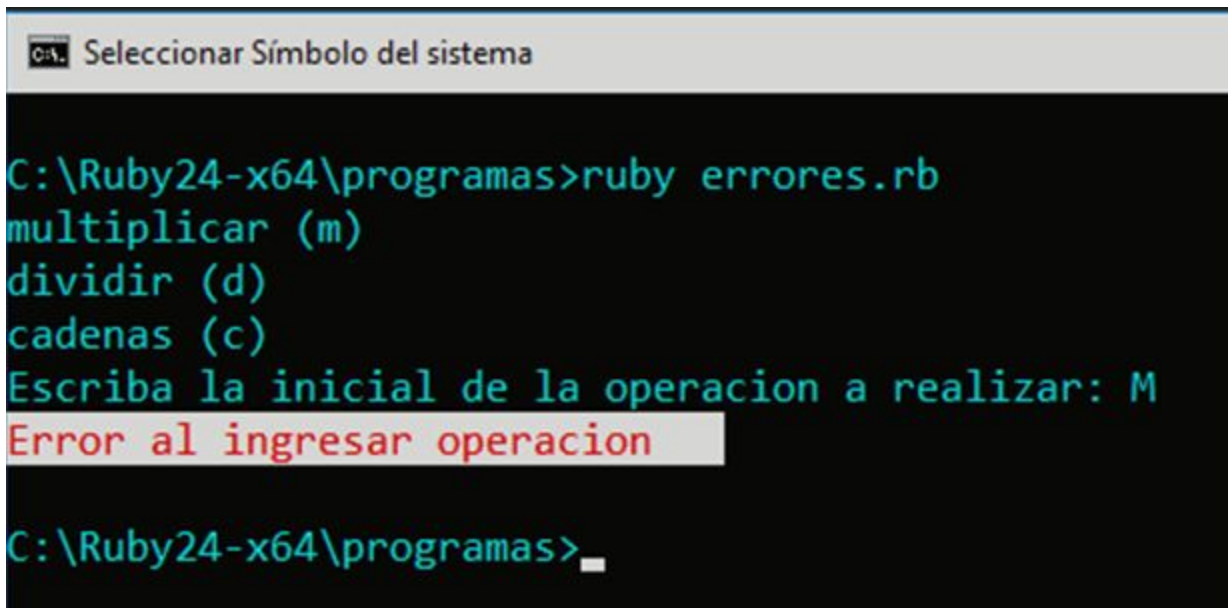
## Sensible a las mayúsculas

El menú está realizado con la ayuda de un case, las opciones se pueden estar declaradas en letras minúsculas.

```
38 when "m"
39     puts "Ingrese dos numeros: "
40     x=gets.to_i
41     y=gets.to_i
42     multiplica(x,y)
43
44 when "d"
45     puts 'Ingrese dos numeros: '
46     x=gets
47     y=gets
48     divide(x,y)
49
50 when "c"
51     puts 'Ingrese una cadena: '
52     cad= gets
53     cadena(cad)
```

Figura 20 - Opciones declaradas en minúsculas.

## Compilación de acceso a las opciones ingresando una letra mayúscula:



```
C:\Ruby24-x64\programas>ruby errores.rb
multiplicar (m)
dividir (d)
cadenas (c)
Escriba la inicial de la operacion a realizar: M
Error al ingresar operacion
C:\Ruby24-x64\programas>
```

Figura 21.- Compilación ingresando una letra mayúscula.

Ruby es sensible a las mayúsculas, eso hace que no podamos realizar la operación deseada si la ingresamos con una letra mayúscula.

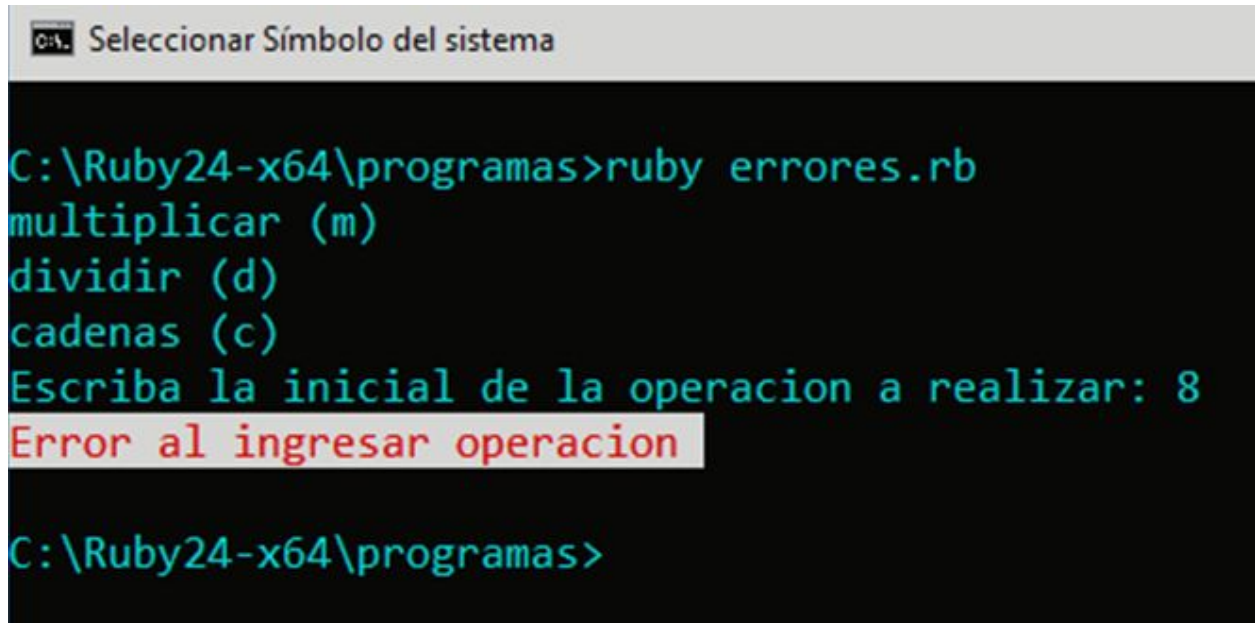
## Ingresar un Dato Erróneo

El menú está realizado con la ayuda de un case, las opciones se pueden estar declaradas en letras minúsculas.

```
38 when "m"
39   puts "Ingrese dos numeros: "
40   x=gets.to_i
41   y=gets.to_i
42   multiplica(x,y)
43
44 when "d"
45   puts 'Ingrese dos numeros: '
46   x=gets
47   y=gets
48   divide(x,y)
49
50 when "c"
51   puts 'Ingrese una cadena: '
52   cad= gets
53   cadena(cad)
```

*Figura 22 - Opciones declaradas en minúsculas.*

Compilación de acceso a las opciones ingresando número:



```
C:\Ruby24-x64\programas>ruby errores.rb
multiplicar (m)
dividir (d)
cadenas (c)
Escriba la inicial de la operacion a realizar: 8
Error al ingresar operacion
C:\Ruby24-x64\programas>
```

*Figura 23 - Compilación ingresando un número.*

Al querer acceder a nuestras opciones ingresando un número es imposible ya que el acceso solo se realiza con letras minúsculas.

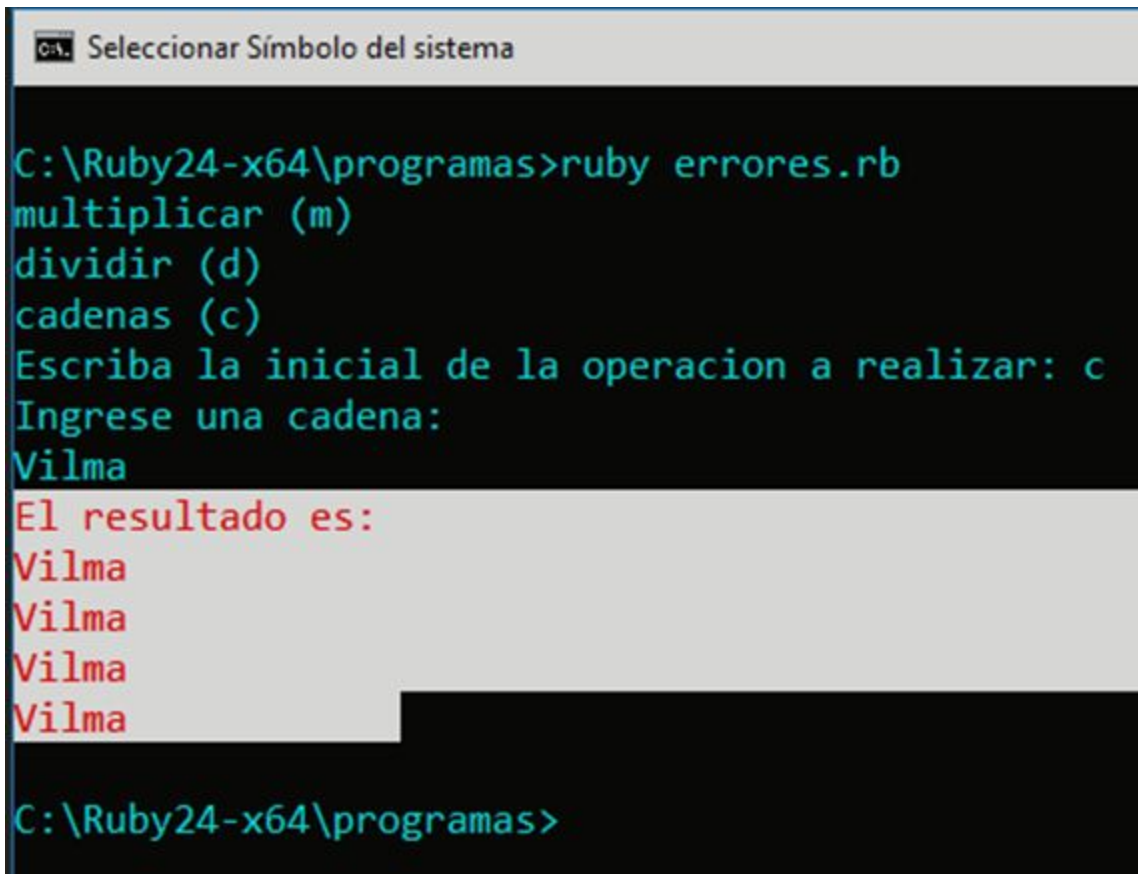
## Multiplicar Cadenas

En la impresión del método cadena tenemos el símbolo \* esta función de Ruby nos ayuda a imprimir varias veces el contenido de la variable de tipo String. En este caso se imprimirá 4 veces el contenido de la variable a.

```
14 #Multiplicar una cadena
15 def cadena(a)
16   puts 'El resultado es: '
17   puts "#{a*4}"
18 end
```

Figura 24 - Uso del símbolo \* en la impresión.

## Compilación del método cadena:



```
C:\Ruby24-x64\programas>ruby errores.rb
multiplicar (m)
dividir (d)
cadenas (c)
Escriba la inicial de la operacion a realizar: c
Ingrese una cadena:
Vilma
El resultado es:
Vilma
Vilma
Vilma
Vilma
C:\Ruby24-x64\programas>
```

Figura 25 -Compilación Impresión múltiple de la cadena.

En Ruby es posible realizar impresiones múltiples del contenido de una variable tipo String con el uso del símbolo \*

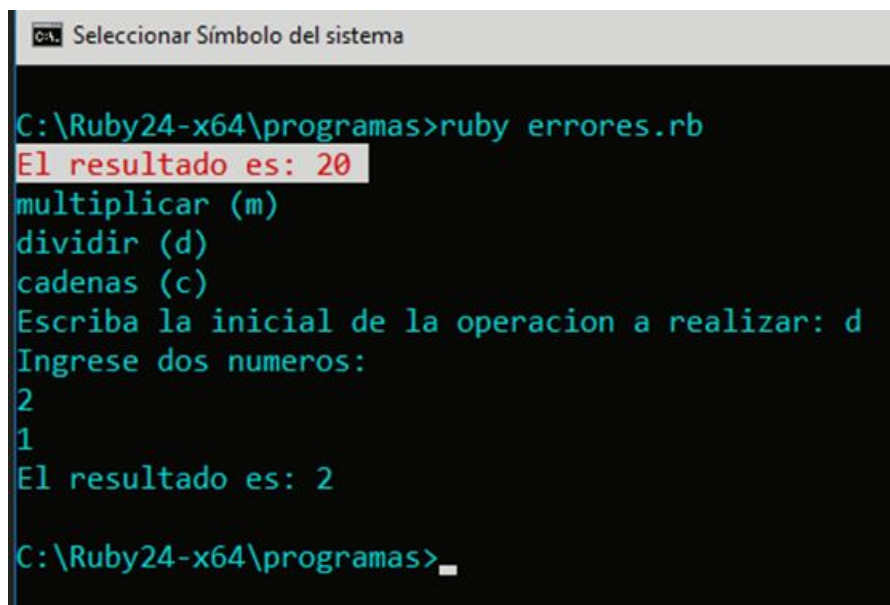
## Función de Parámetros Faltantes

Llamamos al método multiplica directamente en el código pasando como parámetros 10 y 2.

```
4
5 def multiplica(a,b)
6 #Para el uso de puts podemos utili
7   puts "El resultado es: #{a*b}"
8 end
9
10 #Error de Parametros Faltantes
11 multiplica(10,2)
```

Figura 26 - Llamada al método multiplica.

## Compilación de llamada al método multiplica:



```
C:\Ruby24-x64\programas>ruby errores.rb
El resultado es: 20
multiplicar (m)
dividir (d)
cadenas (c)
Escriba la inicial de la operacion a realizar: d
Ingrese dos numeros:
2
1
El resultado es: 2
C:\Ruby24-x64\programas>_
```

Figura 27.- Compilación ingresando una letra mayúscula.

El error semántico se encuentra en la llamada al método multiplica, ya que no debería de realizar otra operación.

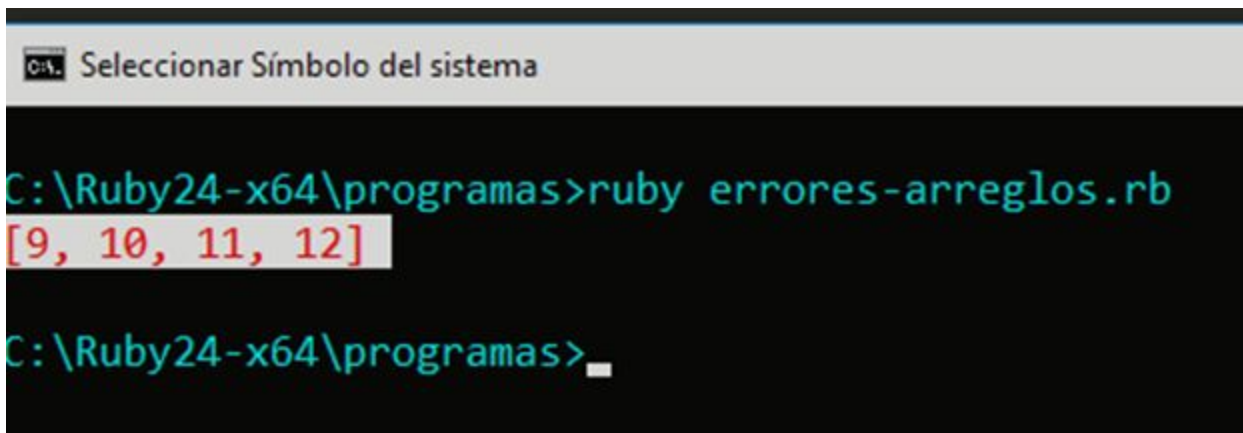
## Arreglos con el mismo nombre

Declaración de tres arreglos con valores distintos, pero dos arreglos han sido declarados con el mismo nombre "vec2".

```
4 vec=[3,4,5,6]
5 vec2=[1,2,vec,7,8]
6 vec2=[9,10,11,12]
7 print vec2
```

Figura 28 - Duplicidad del arreglo vec2.

## Compilación de impresión del arreglo duplicado vec2:



```
Seleccionar Símbolo del sistema

C:\Ruby24-x64\programas>ruby errores-arreglos.rb
[9, 10, 11, 12]

C:\Ruby24-x64\programas>_
```

Figura 29 - Compilación de la impresión del arreglo duplicado vec2.





```
4 vec=[3,4,5,6]
5 vec2=[1,2,vec,7,8]
6 vec3=[9,10,11,12]
7 print vec2
```

*Figura 30 - Declaración de tres arreglos.*

## Compilación de impresión del arreglo vec2:

Seleccionar Símbolo del sistema

```
C:\Ruby24-x64\programas>ruby errores-arreglos.rb  
[1, 2, [3, 4, 5, 6], 7, 8]  
  
C:\Ruby24-x64\programas>_
```

Figura 31 - Compilación de la impresión del arreglo vec2.

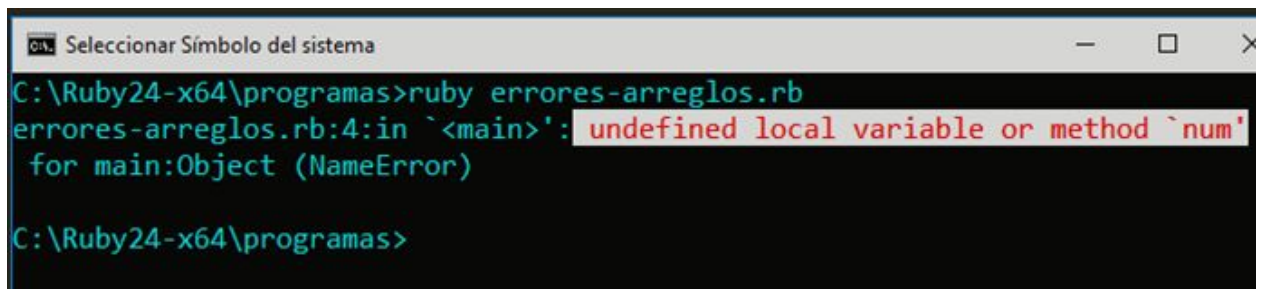
## Variable no Inicializada

Declaración de tres arreglos con valores distintos, en el arreglo con el nombre vec se guarda el contenido de la variable num (La variable num no está inicializada).

```
2 #Variable num no inicilizada
3 #num=3
4 vec=[num,4,5,6]
5 vec2=[1,2,vec,7,8]
6 vec3=[9,10,11,12]
7 print vec2
8
```

Figura 32 - Variable num no inicializada.

## Compilación asignación de la variable num al arreglo vec:



```
Seleccionar Símbolo del sistema
C:\Ruby24-x64\programas>ruby errores-arreglos.rb
errores-arreglos.rb:4:in `<main>': undefined local variable or method `num'
for main:Object (NameError)
C:\Ruby24-x64\programas>
```

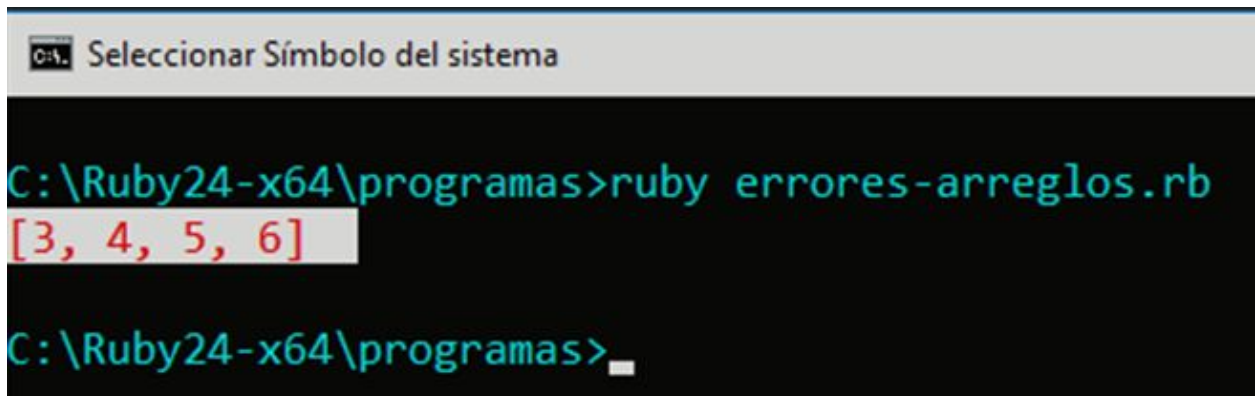
Figura 33 - Compilación de la asignación de la variable num al arreglo vec.

Declaramos la variable num y le asignamos un valor numero en este caso le asignaremos un valor de 3.

```
3 num=3
4 vec=[num,4,5,6]
5 vec2=[1,2,vec,7,8]
6 vec3=[9,10,11,12]
```

*Figura 34 - Declaración y asignación de la variable num.*

## Compilación impresión del arreglo vec:



```
C:\Ruby24-x64\programas>ruby errores-arreglos.rb
[3, 4, 5, 6]
C:\Ruby24-x64\programas>_
```

*Figura 35 - Compilación impresión del arreglo vec.*

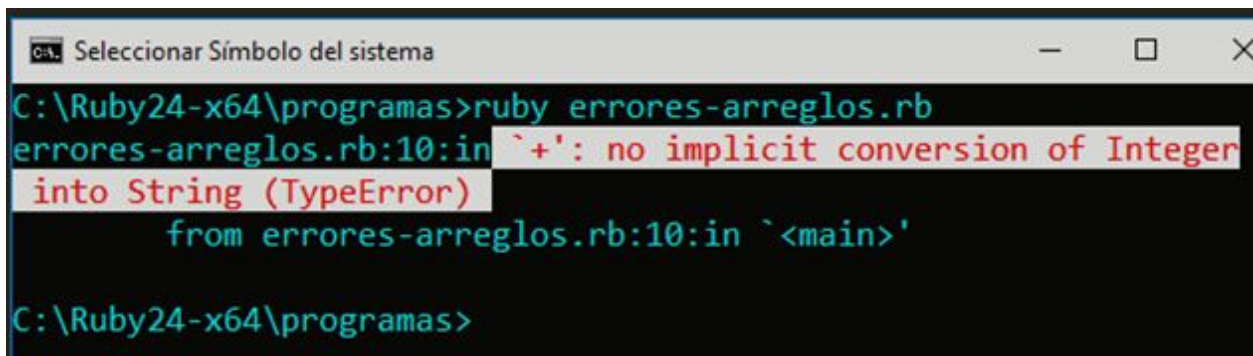
## suma usando un string y un entero

Impresión de una suma usando una cadena (con el contenido de “mensaje”) y un valor numérico (2).

```
9 #Error de suma con string y un entero
10 puts "El resultado es: #{\"mensaje\"+2}"
11 gets()
```

Figura 36 - Suma usando una cadena y un valor numérico.

## Compilación de la suma de una cadena y un valor numérico:



The screenshot shows a Windows command prompt window titled "Seleccionar Símbolo del sistema". The command prompt shows the directory `C:\Ruby24-x64\programas` and the command `ruby errores-arreglos.rb` being executed. The output shows a runtime error: `errores-arreglos.rb:10:in `+': no implicit conversion of Integer into String (TypeError)`. The error message is highlighted in red. The prompt then shows `from errores-arreglos.rb:10:in `<main>'` and the command prompt returns to `C:\Ruby24-x64\programas>`.

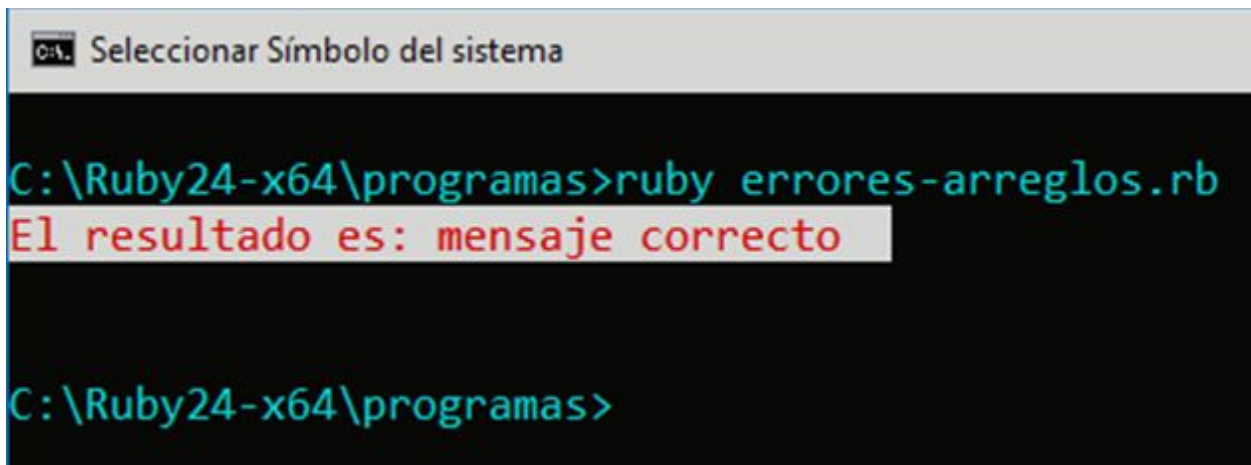
Figura 37 - Compilación de la suma usando una cadena y un valor numérico.

La impresión no se realiza por que en el código realizamos una suma de distintos tipos de datos, a continuación modificamos la impresión sumandos dos valores de tipo string.

```
10 puts "El resultado es: #{\"mensaje\"+\" correcto\"}"
11 gets()
```

*Figura 38 - suma de dos cadenas.*

## Compilación de la suma de dos cadenas:



```
C:\Ruby24-x64\programas>ruby errores-arreglos.rb
El resultado es: mensaje correcto

C:\Ruby24-x64\programas>
```

*Figura 39 - Compilación de la suma usando dos cadenas.*

## Fuente jj

El diseño para implementar el análisis semántico es un poco más complejo que en otros lenguajes más estrictos con la gramática, Ruby no necesita especificar un tipo de dato cuando se declaran variables, lo adquiere cuando se le asigna el primer valor de manera automática.

La lista de tokens **listValues** juega un papel importante, ya que durante el análisis del código fuente, se almacenarán todos aquellos valores que se hayan asignado en alguna declaración en el programa.

Cuando el analizador se encuentra con una declaración, primero guarda el valor en la lista de tokens **listValues**. Si el valor sólo tiene un término, es fácil entender el algoritmo que se implementa para el análisis semántico. Por ejemplo: `num = 2`. El número 2 será reconocido por la expresión regular, cuyo ID es `<VAL_ENTERO>` y como acción semántica, lo agrega a la lista.

```
try{
  (
    id = <ID_LOCAL>
    {
      sm.checkVariable( id );
      listValues.add( id );
    }
    | id = <VAL_ENTERO>
    | id = <VAL_CADENA>
    | id = <VAL_DECIMAL>
    | id = <VAL_CADENA2>
    | "(" valor() ")"
  )
}
```

Figura 40 - Acciones semánticas para almacenar tipos de valores.

Si en la asignación hubiera más de un término, se tendrían que comparar entre cada uno para asegurar que todos son del mismo tipo, pero en el ejemplo sólo es uno, el algoritmo sólo lo compara así mismo y guarda el tipo en la variable **type**.

La función **addVariable()** almacena el nombre de la variable que está contenida en **id** y el tipo que se está contenido en **type**. Finalmente para continuar con el análisis es necesario limpiar la lista de tokens **listValues** para cuando el analizador se vuelva a encontrar con una nueva declaración, y hacer de nuevo el proceso.

```
try {
    ( id = varname()
      <OAS_ASIGNA>
      (
          LOOKAHEAD(5) rangos() <S_SALTO>
          LOOKAHEAD(5) arreglos() <S_SALTO>
          LOOKAHEAD(5) lectura() <S_SALTO>
          LOOKAHEAD(5) parsear()
          valor() <S_SALTO> { for( int i = 0; i < listValues.size(); i++ ) |
                          {
                              for (int j = 0; j < listValues.size(); j++ )
                              {
                                  type = sm.compareTypes( listValues.get( i ), listValues.get( j ) );
                              }
                          }
                          sm.addVariable( id, type );
                          listValues.clear();
                      }
      )
    )
}
```

Figura 41 - Acción semántica para añadir nuevas variables y comparar las existentes.

## Implementación de un error semántico

### *Declaración de tipos de datos asociados a un identificador que no esté duplicado*

Este tipo de error semántico se refiere a qué una vez que se haya definido un tipo de dato (automático) a un identificador, este identificador no pueda volverse a usar, si se le asigna un tipo de dato diferente. Por ejemplo: `num = 0` y más adelante se utilice `num = "Hola"`. El reto de implementar este tipo de error semántico es que el identificador si puede volver a tomar valores nuevos, si son del mismo tipo.

Afortunadamente, este tipo de validación se puede realizar justo antes de añadir el par ordenado (token, type) al mapa de variables; este contiene todos los pares ordenados de todas las declaraciones que se hayan hecho en el código fuente.

El algoritmo consiste en verificar si existe en el mapa de variables, algun identificador que lleve el mismo nombre que el token que está apunto de almacenar, si no es así, simplemente lo almacena. Pero si este token existe en el mapa de variables, se procederá a validar si es del mismo tipo. En caso de ser diferente, se lanza una excepción, en caso contrario se mantiene el mismo identificador en el mapa de variables, haciendo referencia a que el mismo identificador puede tomar diversos valores, si son del mismo tipo por supuesto.

```
public static void addVariable( Token token, String type ) throws SemanticException {  
    if ( (Boolean)mapaVariables.containsKey( token.image ) ){  
        String t = mapaVariables.get(token.image);  
        if (!compareTypes2( type, t )) {  
            throw new SemanticException( SemanticError.DIFFERENT_DECLARATED, token.image, token.beginLine );  
        }  
        else  
            mapaVariables.put( token.image, type );  
    }  
    else{  
        mapaVariables.put( token.image, type );  
    }  
}
```

Figura 42 - Algoritmo para añadir pares ordenados y aplicar validación semántica.



# Instalación

## Requerimientos

Puede instalarse en un equipo propio. Los requisitos mínimos para ejecutar MCVill Compiler son:

| Característica    | Requerimiento               |
|-------------------|-----------------------------|
| Memoria           | 1 Gb                        |
| Procesador        | Doble núcleo, 1GHz          |
| Almacenamiento    | 100 Gb                      |
| Sistema Operativo | Windows 7, 8, 8.1, 10       |
| Terceros          | Java VM (Cualquier version) |

Tabla 8 .- Requerimientos de Instalación

## Step by step

1. Dentro de la ruta **Ruby\Proyecto\Instalador**, encontrará el archivo instalador de **MCVill Compiler** (MCVill\_Setup.exe). Hacer doble clic para comenzar la instalación.



Figura 43 - Instalador de MCVill Compiler.

2. Marcar la opción “Acepto el acuerdo”. A continuación dar clic en **Siguiente**.

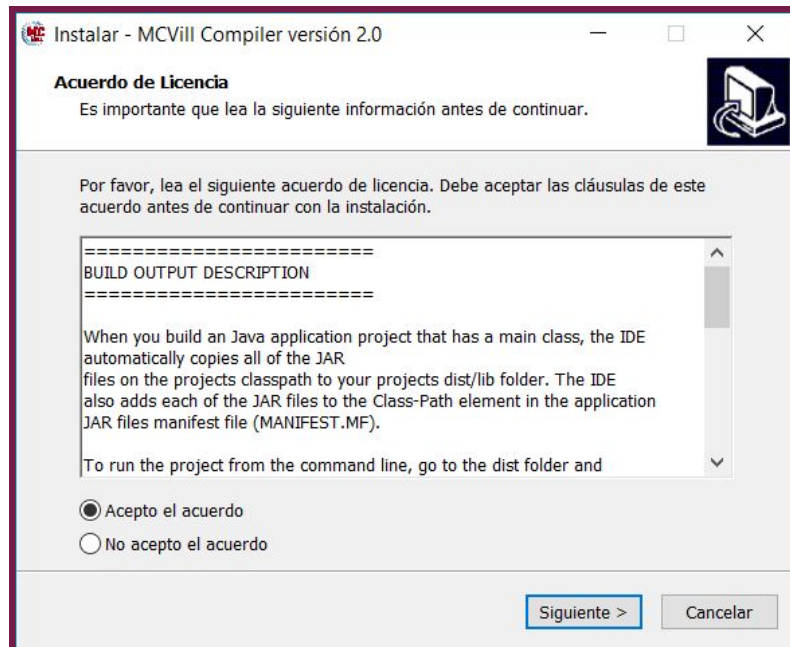


Figura 44 - Acuerdo de Licencia

3. A continuación se muestra información de los requerimientos antes de continuar con la instalación. Dar clic en el botón **Siguiente**.

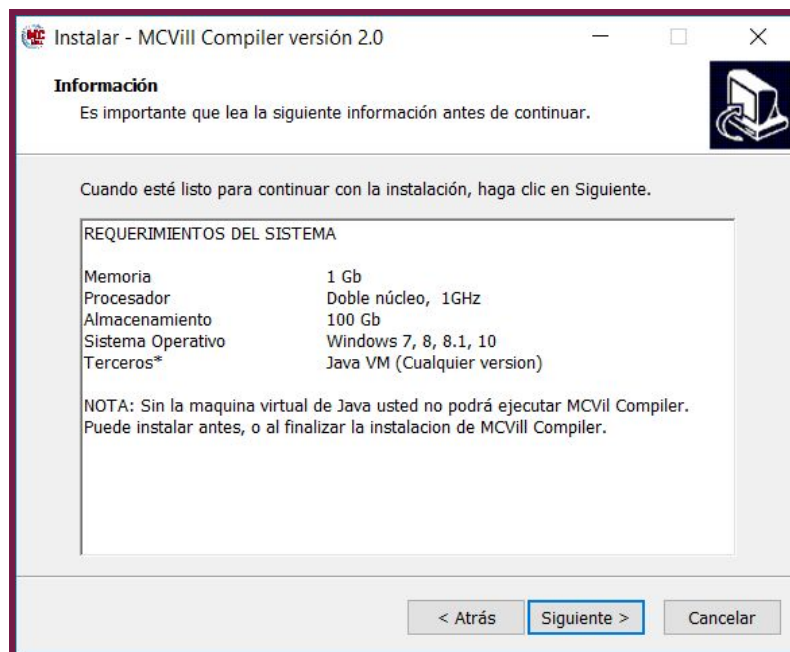
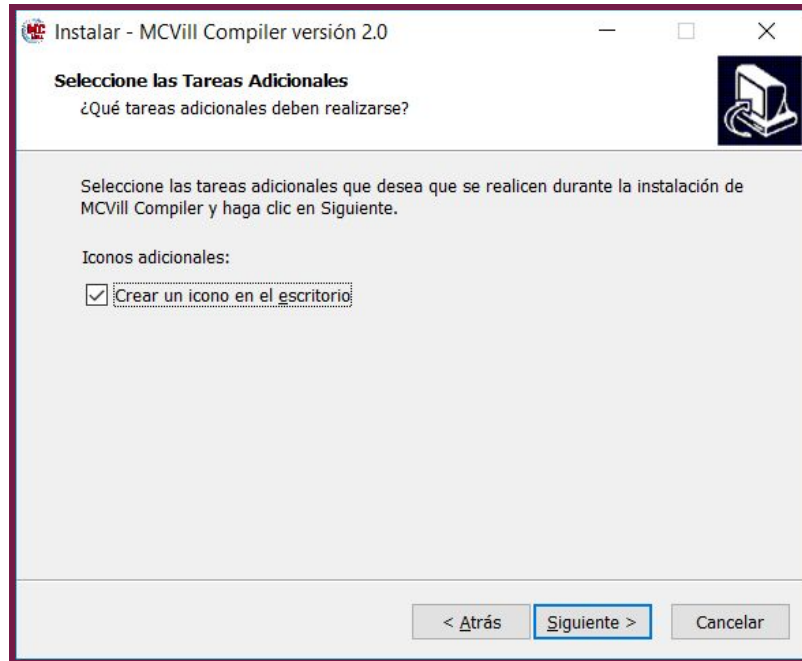


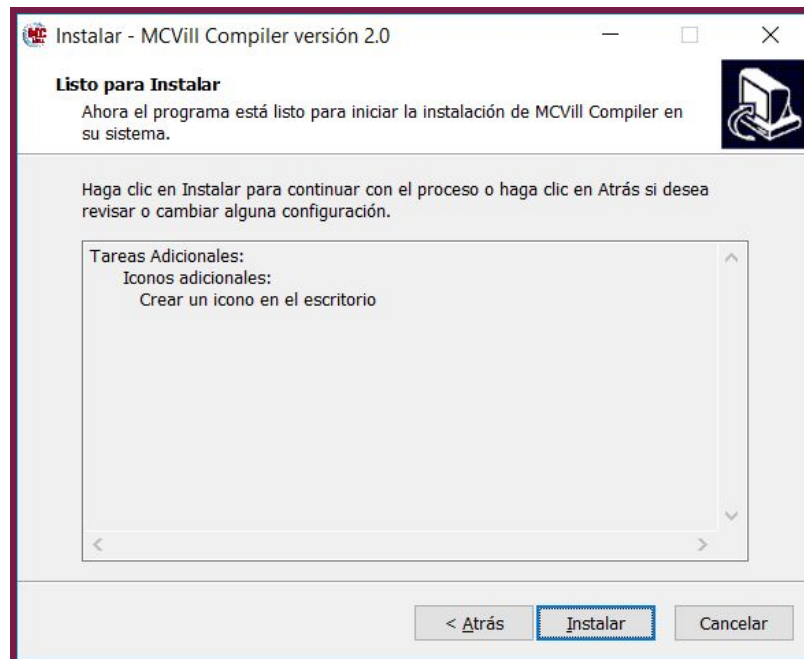
Figura 45 - Información “Requisitos del Sistema”

4. Marcar la casilla “Crear un icono en el escritorio”. Dar clic en **Siguiente**.



*Figura 46 - Tareas Adicionales*

5. Se muestra un panorama general de la instalación. Dar clic en **Instalar**.



*Figura 47 - Panorama General de la Instalación.*

6. Se muestra información antes de finalizar la instalación. Dar clic en el botón **Siguiente** para continuar.

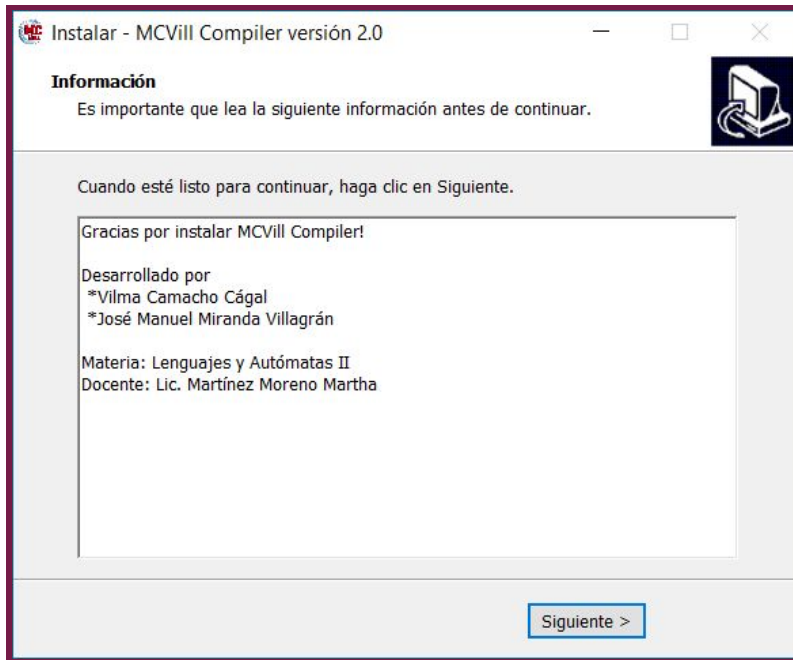


Figura 48 - Información de Instalación de MCVill Compiler.

7. Dar clic en el botón **Finalizar** para concluir la instalación de MCVill Compiler.

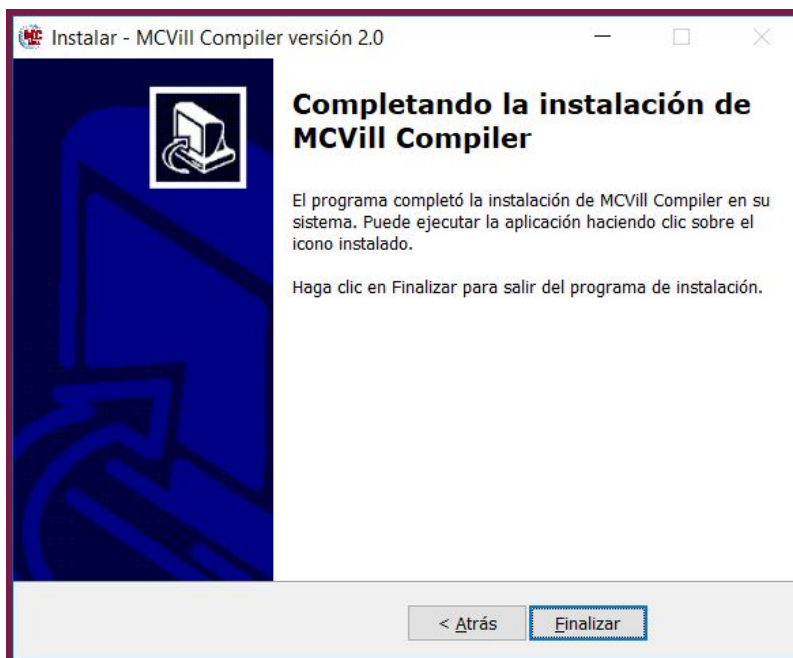
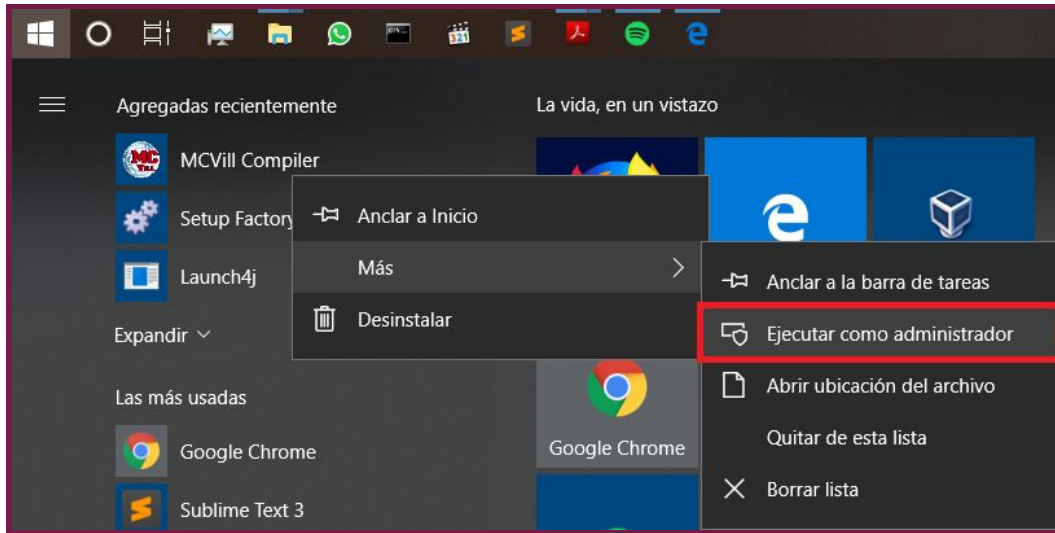


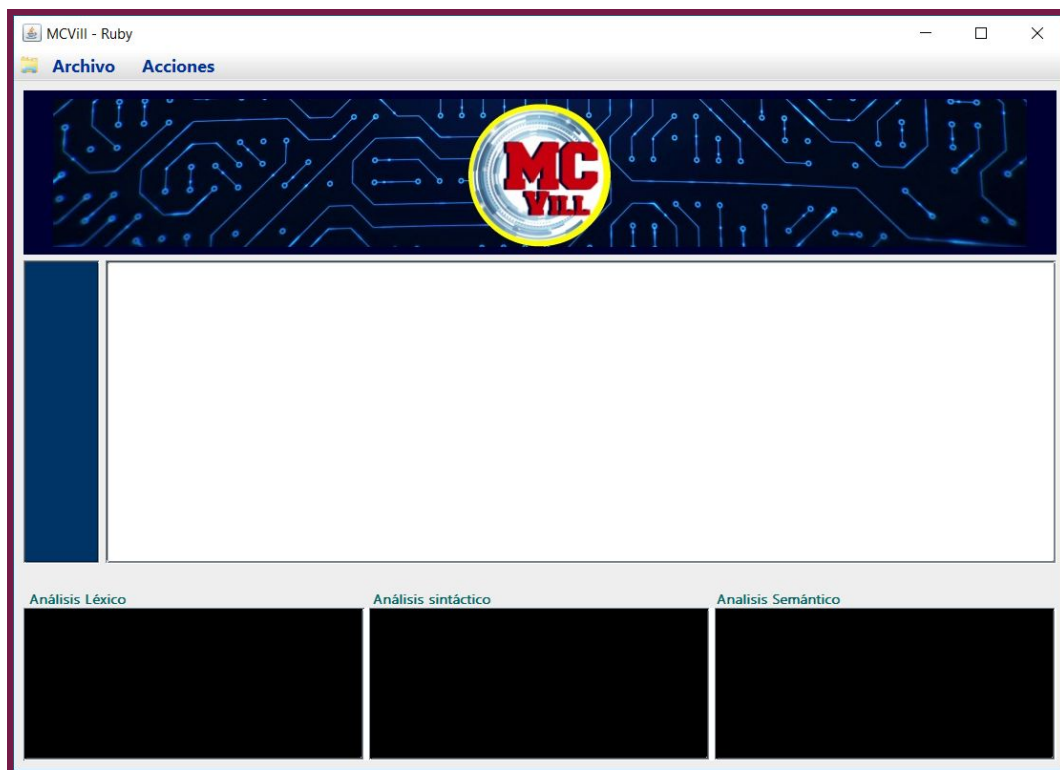
Figura 49 - Instalación Completa.

8. Para iniciar **MCVill Compiler** es necesario *ejecutar como administrador*.



*Figura 50 - Ejecutando MCVill Compiler.*

9. Vista previa de MCVill Compiler. Disfrutalo!



*Figura 51 - Vista Previa de MCVill Compiler.*

# Entorno de edición

## Vista general del compilador MCvill Compiler.

Contamos con:

- Dos menús “Archivo” y “Acciones”
- Un contador de líneas
- Un Panel central para el código fuente
- Tres ventanas de resultado de Análisis
  - “Análisis Léxico”
  - “Análisis Sintáctico”
  - “Análisis Semántico”

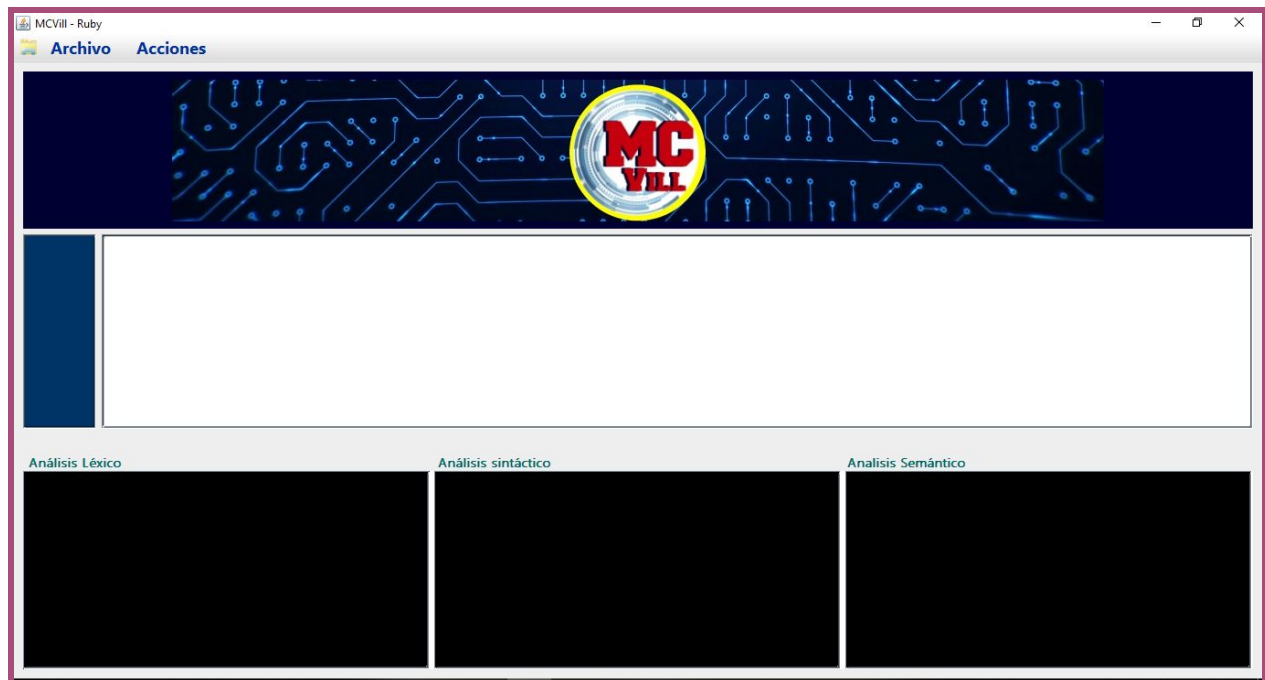


Figura 52 - Vista General de MCvill Compiler.

## Menú "Archivo"

Nos facilita el manejo de archivos compatibles con el compilador MCVill Compiler.



Figura 53 - Menú Archivo

Contamos con los submenús:

## Abrir

Nos permite abrir un archivo compatible con el compilador MCVill Compiler.

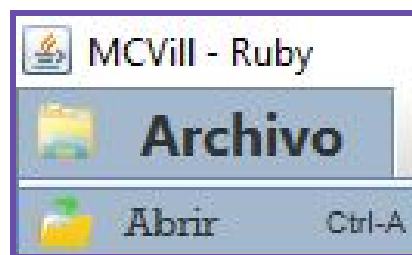
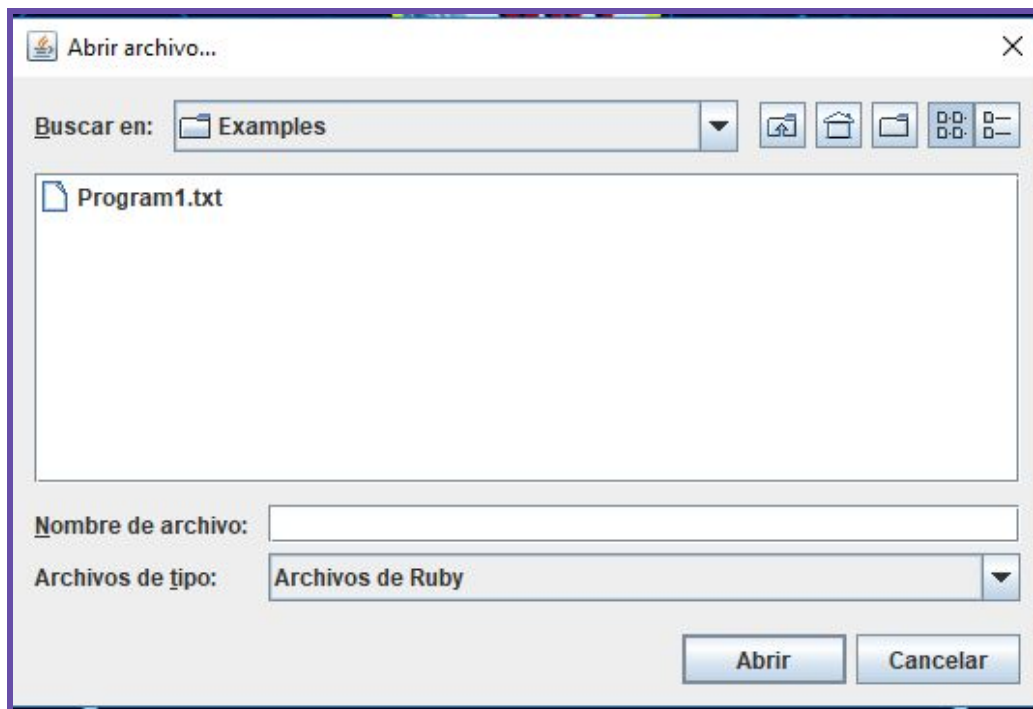


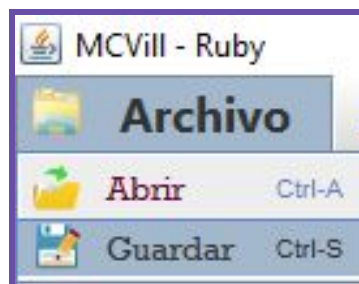
Figura 54 - Submenú "Abrir"



*Figura 55 - Ventana Emergente "Abrir Archivo"*

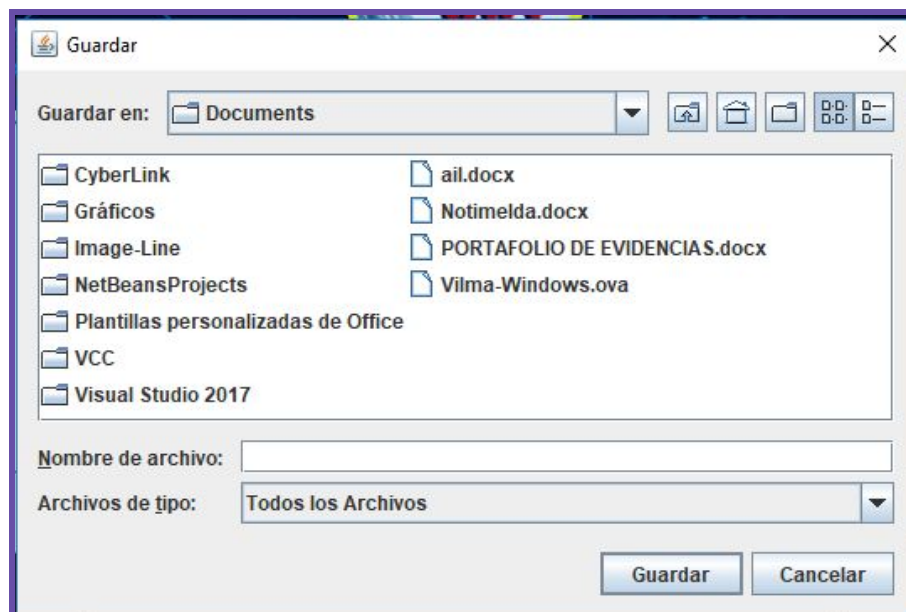
## Guardar

Nos permite Guardar un archivo que se encuentra en el panel de código del MCVill Compiler.



*Figura 56 - Submenú "Guardar"*





*Figura 57 - Ventana Emergente "Guardar"*

## Salir

Nos permite Salir del compilador MCVill Compiler.



*Figura 58 - Submenú "Salir"*

## Menú "Acciones"

Nos facilita realizar diversas acciones en el compilador MCVill Compiler.



Figura 59 - Menú Acciones

Contamos con los submenús:

## Compilar

Nos permite analizar el código que se encuentra en el panel central.



Figura 60 - Submenú "Compilar"

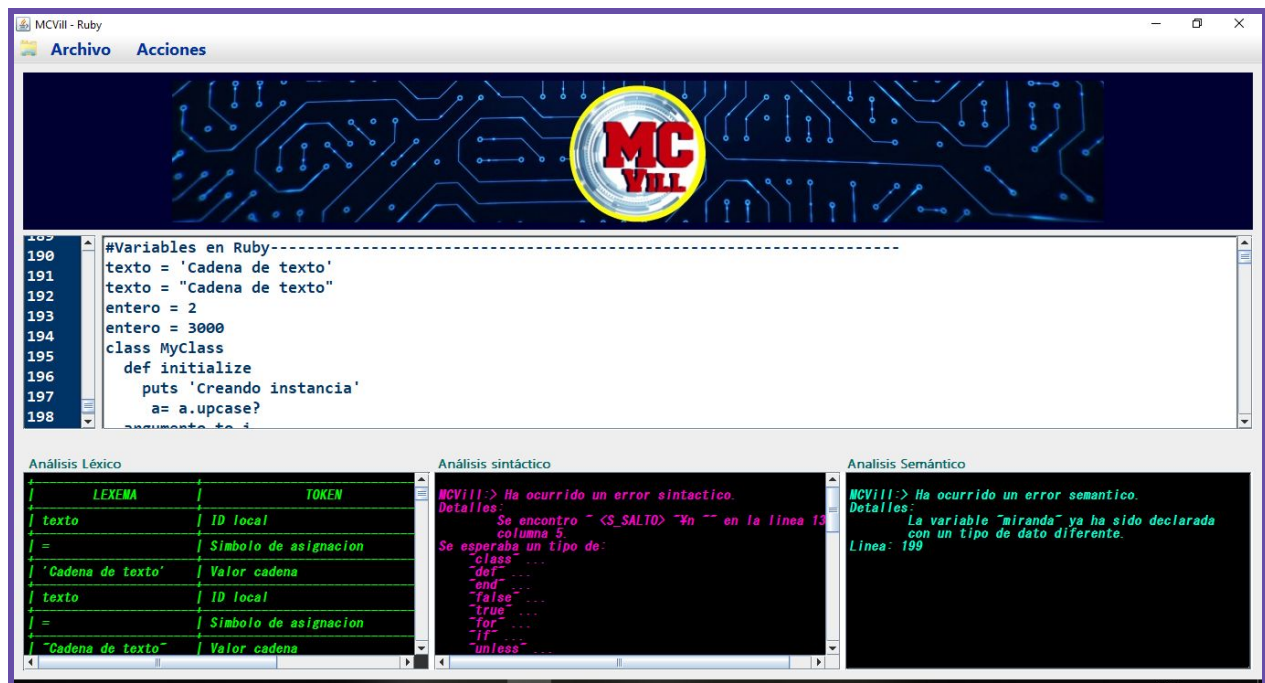


Figura 61 - Vista General al realizar la acción "Compilar"

## Limpiar

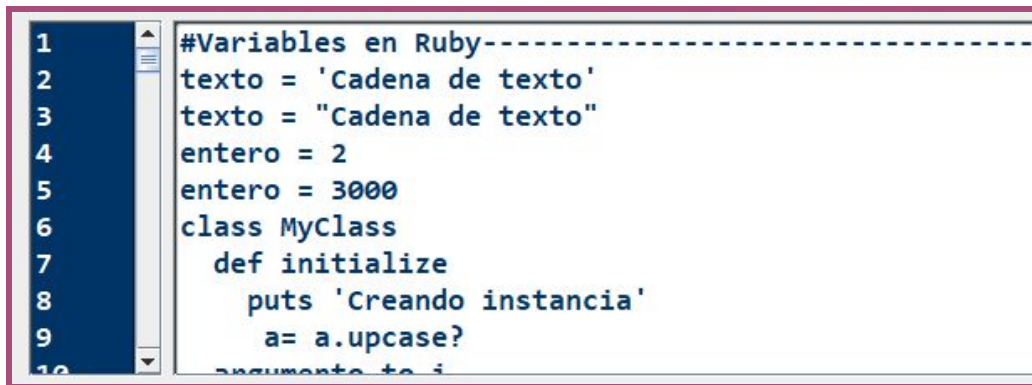
Nos permite borrar el contenido del panel central.



Figura 62 - Submenú "Limpiar"

## Contador de Líneas y Panel Central para el Código Fuente

El contador de líneas nos ayuda a localizar partes específicas del Código que se encuentra en el panel central.

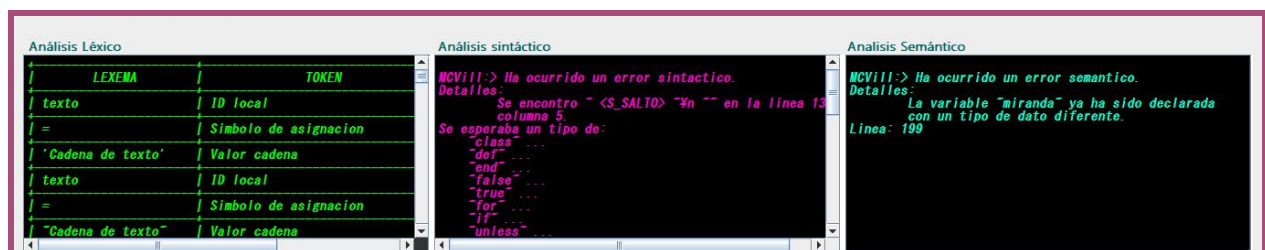


```
1 #Variables en Ruby-----
2 texto = 'Cadena de texto'
3 texto = "Cadena de texto"
4 entero = 2
5 entero = 3000
6 class MyClass
7   def initialize
8     puts 'Creando instancia'
9     a= a.upcase?
10    argumento de i
```

Figura 63 - Contador de Líneas y Panel Central

## Ventanas de Análisis

Nos ayuda a visualizar los resultados encontrados en el análisis de nuestro código fuente.



**Análisis Léxico**

| LEXEMA            | TOKEN                 |
|-------------------|-----------------------|
| texto             | ID local              |
| =                 | Simbolo de asignacion |
| 'Cadena de texto' | Valor cadena          |
| texto             | ID local              |
| =                 | Simbolo de asignacion |
| "Cadena de texto" | Valor cadena          |

**Análisis sintáctico**

```
MCVill-> Ha ocurrido un error sintactico.
Detalles:
Se encontro "<$SALTO>" en la linea 13
columna 5
Se esperaba un tipo de:
class
  def
  end
  raise
  true
  for
  if
  unless
```

**Análisis Semántico**

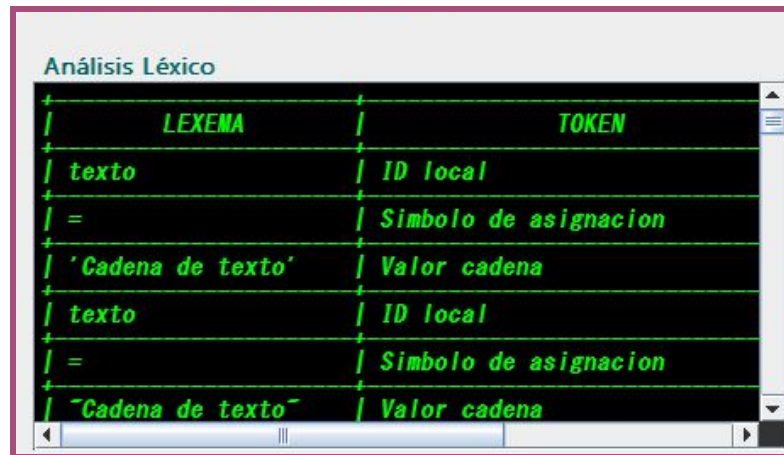
```
MCVill-> Ha ocurrido un error semantico.
Detalles:
La variable "miranda" ya ha sido declarada
con un tipo de dato diferente.
Linea: 199
```

Figura 64 - Ventanas de Análisis

## Ventana del Análisis Léxico

Reconoce que las palabras escritas en el código analizado pertenezcan al lenguaje Ruby.

Nos muestra una salida en forma de tabla con el Lexema encontrado y el tipo de Token.



The screenshot shows a window titled 'Análisis Léxico' containing a table with two columns: 'LEXEMA' and 'TOKEN'. The table lists several tokens from a Ruby code snippet, including identifiers, assignment symbols, and string literals.

| LEXEMA            | TOKEN                 |
|-------------------|-----------------------|
| texto             | ID local              |
| =                 | Simbolo de asignacion |
| 'Cadena de texto' | Valor cadena          |
| texto             | ID local              |
| =                 | Simbolo de asignacion |
| "Cadena de texto" | Valor cadena          |

Figura 65 - Ventana del Análisis Léxico

## Ventana del Análisis Sintáctico

Los **errores sintácticos**, ocurren cuando el programador escribe código que no va de acuerdo a las reglas de escritura del lenguaje de programación.

Nos muestra una salida que nos informa el lexema incorrecto, el lexema que se esperaba encontrar y el numero de linea donde localizamos el error.

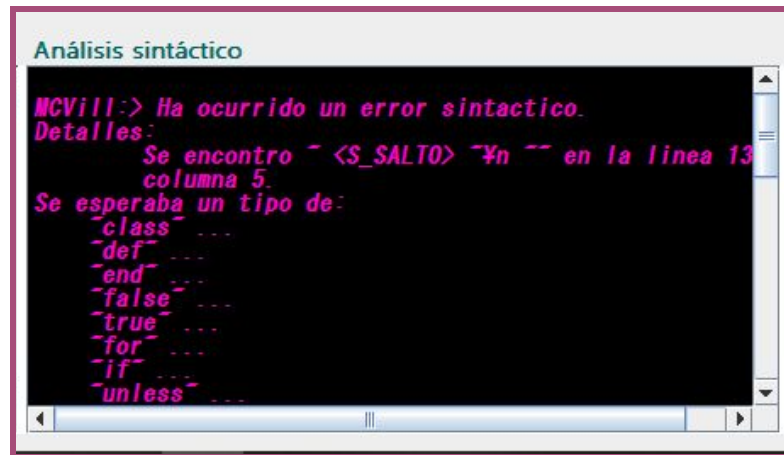


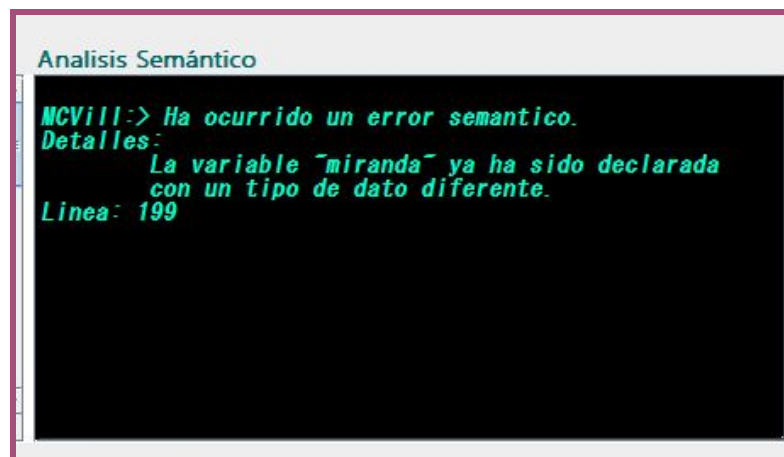
Figura 66 - Ventana del Análisis Sintáctico

## Ventana del Análisis Semántico

Un error semántico se produce cuando la sintaxis del código es correcta, pero la semántica o el significado no es el que se pretendía.

Un error semántico puede hacer que el programa termine de forma anormal, con o sin un mensaje de error.

Nos muestra una salida que nos informa el lexema incorrecto, que tipo de error ocurrió y el número de línea en donde se localiza el error.

A screenshot of a software window titled "Análisis Semántico". The window has a black background with green text. The text displays a semantic error message: "MCVill:> Ha ocurrido un error semantico." followed by "Detalles:" and a detailed message "La variable 'miranda' ya ha sido declarada con un tipo de dato diferente." and "Linea: 199".

```
Analisis Semántico
MCVill:> Ha ocurrido un error semantico.
Detalles:
    La variable "miranda" ya ha sido declarada
    con un tipo de dato diferente.
Linea: 199
```

Figura 67 - Ventana del Análisis Semántico

# Compilación

## Sin error

En el apartado de *definición de métodos en Ruby*, tenemos definido el método **mi\_metodo**, donde sólo se realiza una simple suma de dos enteros e imprime el resultado invocando el método. En la línea 200 está comentada una declaración que simulará un futuro error. MCVill Compiler deja en “blanco” las pantallas de salida de Análisis Sintáctico y Análisis Semántico, indicando que no hubo error en la compilación. Mientras que la pantalla de Analisis Lexico siempre se muestra la tabla de par ordenado.

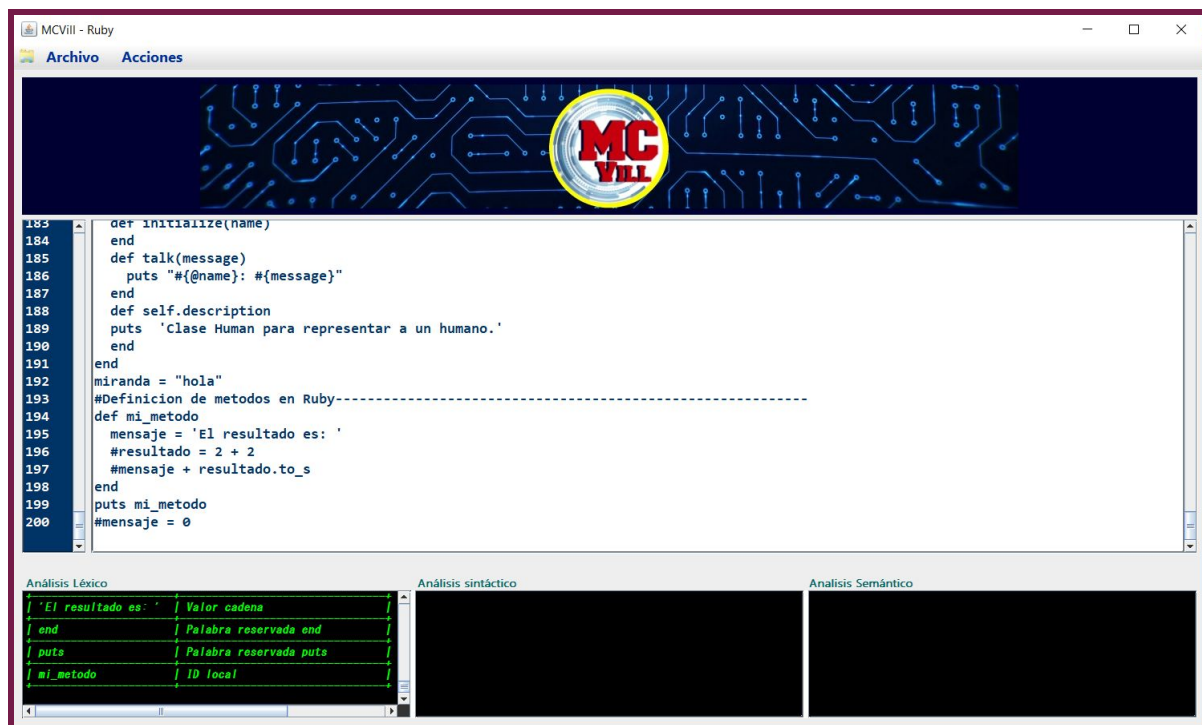


Figura 68 - Compilación sin errores.



## Con error

Retomando el ejemplo anterior del apartado de *Definición de métodos en Ruby* la declaración de la variable **mensaje** (línea 195) es de tipo String. Más adelante (línea 200) se vuelve a declarar la variable **mensaje**, pero se le asigna un valor entero. MCVill Compiler es capaz de detectar este tipo de errores semánticos cuando compilamos. En la pantalla de Análisis Semántico se muestra el error, y además ofrece detalles. Mientras que en la pantalla de Análisis Sintáctico se muestra otro tipo de error; en la línea 201 se encuentra un caracter no valido para el lenguaje, por lo tanto se trata de error lexicográfico.

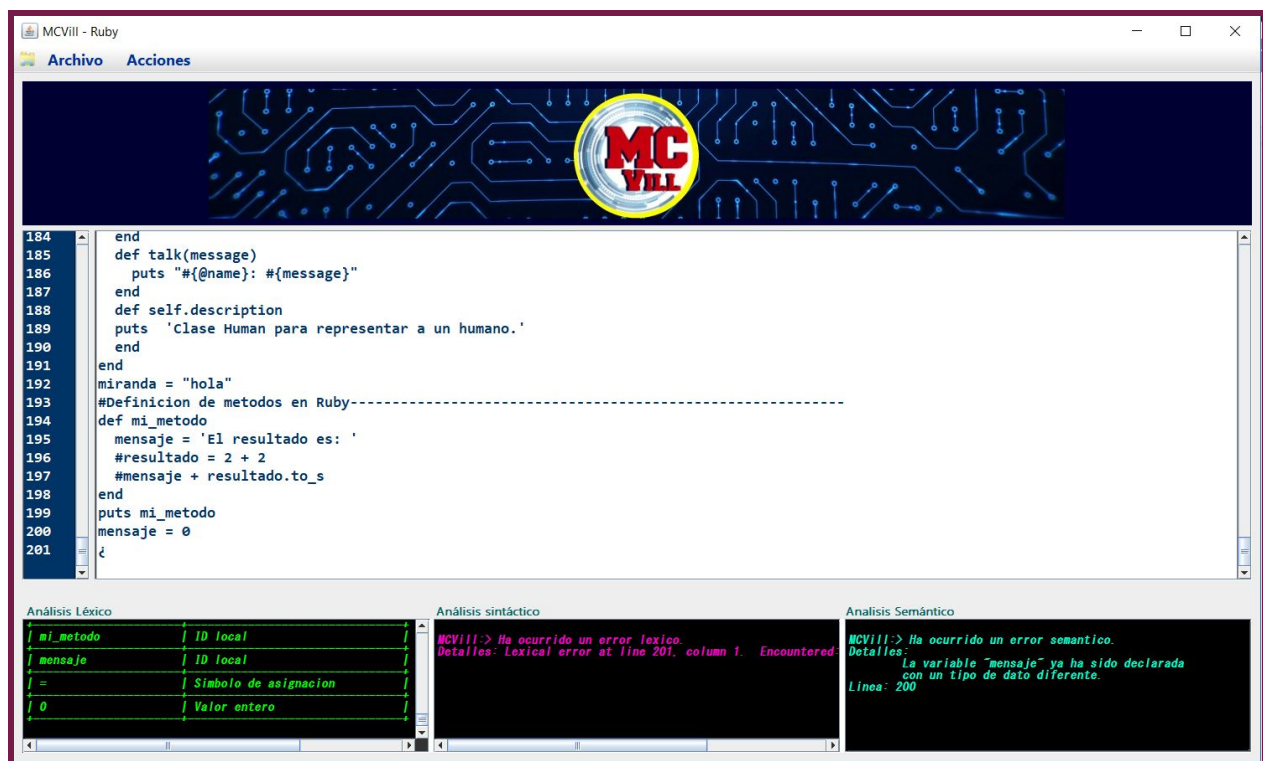


Figura 69 - Compilación con errores.

# Bibliografía

- Irvine, K., Romero Elizondo, A., Villar Cuesta, M., Leal Flores, A. and Morán Loza, J. (2008). *Lenguaje ensamblador para computadoras basadas en Intel®*. 5th ed. México: Pearson Educación.
- Gálvez, S. and Mora Mata, M. (2005). *Java a tope: Traductores y compiladores con LEX/YACC, JFlex/Cup y JavaCC*. 1st ed. Málaga.
- Aho, A., Ullman, J., Sethi, R. and Lam, M. (2008). *Compiladores, principios, técnicas y herramientas*. 2nd ed. México: Pearson Education.
- Ruby-lang.org. (2018). *Documentación*. [online] Available at: <https://www.ruby-lang.org/es/documentation/> [Accessed 15 Nov. 2018].
- <http://es.tldp.org/Manuales-LuCAS/doc-guia-usuario-ruby/guia-usuario-ruby.pdf>