Lenguaje de programación Emerald

Procesadores de Lenguajes

Alejandro Sánchez Medina
Aridane J. Sarrionandia de León
Garoe Dorta Pérez
Mario Lemes Medina
Moisés J. Bonilla Caraballo
Escuela de Ingeniería Informática
Universidad de Las Palmas de Gran Canaria

Índice de contenido

1. Introducción

En las siguientes páginas se presenta la especificación y el analizador sintáctico de un lenguaje de programación al que hemos denominado Emerald. Se trata de un subconjunto de Ruby, un lenguaje interpretado y orientado a objetos, desarrollado por Yukihiro Matsumoto.

Nota: esta memoria forma parte de una entrega acumulativa. Para ver una lista no exhaustiva de los cambios con respecto en entregas anteriores, puede dirigirse al anexo situado al final de este mismo documento.

2. Conceptos fundamentales

2.1. Objetos

Un objeto consiste en un conjunto de variables de instancia más un comportamiento (conjunto de métodos que trabajan sobre el objeto). En Ruby/Emerald, todo valor manejado directamente por el programa es un objeto, como p.e.:

- Un valor referido por una variable.
- El argumento de un método.
- Un valor que es devuelto como resultado de evaluar una expresión, una sentencia o un programa.

2.2. Variables

Una variable es una asociación entre uno o más nombres (o identificadores) y un objeto. Dicho objeto se conoce como valor de la variable. En Emerald diferenciamos entre cuatro tipos de variables: constantes, locales, globales y de instancia.

2.3. Métodos

Procedimiento que se invoca desde un objeto y que trabaja sobre dicho objeto. Un método tiene una visibilidad que define desde dónde puede invocarse.

2.4. Bloques

Método anónimo que se pasa como argumento a un método.

Ejemplo 1: En el siguiente programa, para cada elemento del array, el bloque { |i| puts i } es llamado por el método each de la clase Array.

```
a = [1, 2, 3]
a.each { |i| puts i }
```

2.5. Clases

Se trata de construcciones que agrupan varias variables relacionadas entre sí. Las clases en Emerald no tienen métodos, pareciéndose más a simples estructuras.

Ejemplo de definición de estructura:

```
class Punto
   @x = 0
   @y = 0
end
```

El acceso a las variables se hará sin referirse al @ en la forma:

```
p = Punto.new
p.x = 1
p.y = 2
```

2.6. Valores booleanos

Los objetos se clasifican entre objetos verdaderos y falsos. Sólo false y nil son objetos falsos.

false → única instancia del objeto False → resultado de una expresión-false

nil → única instancia del objeto Nil → resultado de una expresión-nil.

true → única instancia de la clase True → resultado de una expresión-true.

2.7. Recursividad

Ruby/Emerald admite recursividad.

2.8. Entrada/Salida

La salida estándar en Emerald sigue la forma:

puts <string>

ó

puts <expresión>

La entrada sigue el formato:

<variable> = getc

La instrucción anterior hace que se guarde en "<variable>" el valor introducido por el usuario.

3. Estructura léxica

El texto de un programa se convierte en una secuencia de elementos de entrada que pueden ser fines de línea, espacios en blanco, comentarios, marca fin de programa o tokens.

3.1. Texto del programa

source-character :: [any character in ISO/IEC 646:1991 IRV]

3.2. Fin de línea

Se admite el fin de línea usado en GNU/Linux, representado por el carácter 'nueva línea' o \n (código ASCII 10).

3.3. Espacio en blanco

Se acepta como espacio en blanco los caracteres “tabulador” (0x09) y “espacio” (0x20)

3.4. Comentarios

En Emerald únicamente permitimos los comentarios de una línea, que son aquellos precedidos por '#'. Por ejemplo:

a + b # Esto es un comentario.

3.5. Tokens

En este apartado se lista las palabras claves y se describen los identificadores, signos de puntuación, operadores y literales.

3.5.1. Palabras clave

Las palabras clave de Emerald se listan a continuación:

def	end	if	then	else	while
do	class	and	or	not	nil
false	true	.each	.new	Array	

Nota: las palabras clave son case-sensitive.

3.5.2. Identificadores

Los identificadores siguen un formato según el tipo de objeto al que se refieran:

- Variables locales: [a-z_] ([a-zA-Z0-9_])*
- Variables globales: $[a-zA-Z_]([a-zA-Z0-9_])*
- Variables de instancia @[a-zA-Z_]([a-zA-Z0-9_])*
- Constantes: [A-Z] ([A-Z0-9_])*
- Método: [a-z_] ([a-zA-Z0-9_])*

Las variables locales y los métodos no podrán compartir nombre. De esta manera no existe confusión al hacer referencia al nombre para llamar al método o la variable.

3.5.3. Signos de puntuación (punctuators)

Los signos de puntuación admitidos son los siguientes:

[	]	{	}
(	)	,	;

3.5.4. Operadores

Lista de operadores de Emerald:

!=	=	[]	/	==	>	>=
<	<=	<<	>>	+	-	*
+@	-@	[]=				

3.5.5. Literales

En Emerald diferenciamos entre literales numéricos, strings y arrays.

3.5.5.1. Literales numéricos

Siguen el patrón

(+|-)? (entero|float)

donde

entero = (0 | [1-9] ([0-9])*)

y

float = (entero.entero | .entero | entero.) ((e|E)entero)?

Nota: Si el token anterior a un signed-number es un identificador, debe haber al menos un espacio en blanco o un fin de línea entre ambos tokens.

Por ejemplo:

- x -123: hay un espacio en medio, x es el nombre de un procedimiento y -123 un número suministrado como argumento.
- x-123: no hay espacios en medio, se llama al método "-" de x con 123 como argumento.

Un literal numérico puede evaluarse como un entero o como un float.

3.5.5.2. Literales carácter

Los caracteres se expresan entre comillas simples (‘). Por ejemplo:

‘a’, ‘b’, ‘c’, ‘\n’

son literales carácter. Los literales carácter admiten secuencias de escape (ver apartado 3.5.5.4), y se almacenan en memoria mediante su código ASCII.

3.5.5.3. Literales string

Se trata de secuencias de caracteres que van encerrados entre comillas dobles (“).

La string admite secuencias de escape (ver siguiente apartado) e interpolación (cadenas de la forma #{<variable>} que en la salida se sustituyen por el valor de la variable dada).

Por ejemplo:

a = 6

puts "La suma es igual a #{a}."

Daría como salida:

La suma es igual a 6.

3.5.5.4. Secuencias de escape

Las secuencias de escape admitidas en un literal carácter o string son las siguientes:

Secuencia	Significado	Secuencia	Significado
\\	\	\t	Tabulador
\”	Comillas dobles	\n	Salto de línea
\'	Comillas simples (*)		

(*) Para literales carácter.

4. Ámbito de las variables

El ámbito de una variable es la región o conjunto de regiones de un programa en los que dicha variable es accesible. Dicho ámbito depende del tipo de variable, que en Emerald puede ser constante, de instancia, local o global.

4.1. Ámbito de las constantes y las variables de instancia

Las constantes y las variables de instancia no tienen ámbito; su “visibilidad” depende del contexto de ejecución actual.

4.2. Ámbito de las variables locales

Una variable local es únicamente accesible desde el cuerpo del método en el que se declaró. Cuando se declara una variable local a la función main se produce un caso especial, pues como como se verá en el apartado 5, no existe un main explícito. Esto podría llevarnos a pensar que, en este caso, no existe diferencia semántica entre declarar una variable local al main y una variable global. Sin embargo, si existe tal diferencia: una variable local al main no es accesible desde los métodos definidos dentro de este, mientras que una variable global sí lo es.

4.3. Ámbito de las variables globales

Las variables globales son accesibles desde cualquier parte del programa.

5. Estructura de un programa

Un programa en Emerald consiste en una secuencia de instrucciones con definiciones (de métodos, de clases, etc) que pueden estar al principio, en medio o al final del código fuente; no existe un main explícito. Por ejemplo, lo siguiente es un programa Emerald perfectamente válido:

```
a = 5

b = 6

def Suma (x, y)

   x + y

end

Suma( a, b)
```

En el caso anterior, el código dentro de la función Suma no se ejecuta hasta que dicha función es invocada.

5.1. Return implícito

En Emerald no existe una instrucción return; el valor devuelto por un método o programa es el resultado de su última instrucción.

6. Expresiones

6.1. Expresiones lógicas

Se dispone de las palabras clave not, and y or para realizar las operaciones correspondientes que se listan a continuación:

- not <exp> → Niega el resultado de evaluar la expresión <exp>
- <exp_1> and <exp_2> → Devuelve el resultado de aplicar un “y lógico” a los resultados de las expresiones <exp_1> y <exp_2>.
- <exp_1> or <exp_2> → Devuelve el resultado de aplicar un “o lógico” a los resultados de las expresiones <exp_1> y <exp_2>.

El orden de precedencia es: not > and > or.

6.2. Metodos

6.2.1. Invocación de método

Los métodos se invocan siguiendo el siguiente prototipo:

nombre_método [argumentos] [bloque]

Donde argumentos es una lista de argumentos separados por comas y encerrada o no entre paréntesis.

Por ejemplo:

suma(1, 2)

y

suma 1, 2

son ambos válidos. Si el resultado de la invocación se va a usar en el mismo instante en que se recibe, es obligatorio usar paréntesis. Por ejemplo:

suma(1, 2).inverso

[bloque] se refiere a un bloque de Ruby/Emerald, un concepto que ya se introdujo en el apartado 2.4 y en el que se ahondará en el siguiente apartado. El paso de bloques como parámetros sólo está definido para una serie de funciones definidas de antemano. El usuario no puede crear métodos con bloques como parámetros.

6.2.2. Definición de métodos

Para definir un método se sigue el siguiente patrón:

def <nombre_método> (<lista_argumentos>)?

(<cuerpo_método>)?

end

<nombre_método> es el identificador del método. <lista_argumentos> es una lista de argumentos separados por comas (,) y encerrada o no entre paréntesis. <cuerpo_método> es el código interno del método.

Nota: dentro de <cuerpo_método> no se permite definir clases ni asignar valores a constantes.

6.2.3. Parámetros de métodos

En Emerald sólo se permiten los parámetros obligatorios. Por cada parámetro obligatorio indicado en la definición del método, ha de existir un argumento correspondiente en la invocación del método.

6.2.4. Visibilidad de métodos

Todos los métodos de Emerald son públicos.

6.3. Bloques

Como ya se comentó anteriormente, un bloque es un bloque de código (o método anónimo) que se le pasa como parámetro a un método o función. Veamos el concepto con el ejemplo de una llamada a un método:

```
array = [1, 2, 3, 4]

array.collect do |n|

  n ** 2

end



10.times do |i|

    puts i

    j = 4

end

puts j

puts i
```

Tanto puts j como puts i devuelve nil, porque ambas variables solo son accesibles desde el ámbito del bloque. En definitiva un bloque se comporta como un método.

Lo anterior es una llamada al método collect de la clase Array. Dicho método recorre el array y, para cada elemento, ejecuta el bloque de código suministrado como parámetro, en este caso:

```
do |n|

  n ** 2

end
```

|n| es una variable de bloque, que en este caso, se rellena automáticamente en cada iteración con el elemento actual del array.

Si el bloque que definimos es pequeño, una versión alternativa consiste en encerrarlo entre llaves y en una única línea:

```
array.collect! { |n] n ** 2 }
```

6.4. Operadores

Se distinguen tres tipos: asignaciones, operadores unitarios y operadores binarios.

6.4.1. Asignaciones

Son del tipo:

<variable> = <expresión asignable>

Donde <variable> puede ser:

- Variables simples
- Variables de estructura
- Un elemento de un array

Por otro lado, <expresión asignable> puede ser una de las siguientes variantes:

- Variables simples
- Variables de estructura
- Arrays
- Literales
- <expresión asignable> (operador) <operando>

Donde los operadores pueden ser todos los definidos anteriormente y operando puede ser una expresión asignable exceptuando a los Arrays y Literales.

6.4.2. Operadores unarios

Se trata de los operadores +, – (opuesto) y not(negación), aplicables a enteros y números en coma flotante.

6.4.3. Operadores binarios

Ruby admite los siguientes operadores binarios, aplicables a enteros y números en coma flotante.

6.4.3.1. Operadores aritméticos

Operador	Significado	Operador	Significado
+	Suma	*	Multiplicación
-	Resta	/	División

6.4.3.2. Operadores de comparación

Operador	Significado	Operador	Significado
==	Igualdad	>	Mayor que
!=	Desigualdad	>=	Mayor o igual que
<	Menor que	<=	Menor o igual que

6.5. Expresiones primarias

Se aceptan las siguientes: definición de clase, definición de método, construcciones if y while, expresiones, literales e invocación de métodos.

6.5.1. Estructuras de control

Dichas estructuras cambian el flujo de ejecución del programa. Se clasifican en expresiones condicionales y bucles.

6.5.1.1. Expresiones condicionales

Incluye las expresiones if.

6.5.1.1.1. Expresión if

Sigue la forma:

if <condición> (**then** | <salto de línea>)?

<sentencias>

(**else** (**then** | <salto de línea>)?)?

end

6.5.1.2. Bucles

En Emerald sólo se define el bucle while.

6.5.1.2.1. Bucle while

Sigue el patrón:

while <condición> **do**

<código>

end

Ejecuta <código> mientras se cumpla la condición <condición>

6.5.2. Agrupando expresiones

Para ello basta con poner entre paréntesis la expresión que queramos agrupar.

6.5.3. Referencias a variables

- Constantes
- Variables globales
- Variables de instancia
- Variables locales

6.5.3.1. Pseudovariables

Se refiere a los valores false y true, literales de boolean y nil.

7. Clases

Tal como se dijo en el apartado 2: las clases son construcciones que agrupan varias variables relacionadas entre sí. Las clases en Emerald no tienen métodos, pareciéndose más a simples estructuras. Los atributos solo pertenecerán a las clases simples integer, float, char o boolean.

Ejemplo de definición de estructura:

```
class Punto
    @x = 0
    @y = 0
end
```

Recordatorio: para acceder a los campos de la clase obviamos el @ del identificador. Por ejemplo: para acceder al campo x de la instancia P lo haríamos mediante P.x

7.1. Creación de instancias

Una instancia puede ser creada llamando al método new de la clase, teniendo el siguiente comportamiento:

- Crea una instancia directa del receptor sin enlaces a las variables de instancia.

7.2. Clases predefinidas de Emerald

7.2.1. NilClass

Esta clase sólo tiene una instancia, nil, y no permite la creación de otras instancias. La instancia nil tiene las siguientes propiedades:

- nil and <expresión> devuelve siempre false.
- nil or <expresión> devuelve el valor de <expresión>

7.2.2. Boolean

Esta clase solo tiene dos instancias, true y false. Además no se pueden crear instancias de la clase Boolean La instancia true goza de las siguientes propiedades:

- true **and** <expresión> devuelve el valor de <expresión>
- true **or** <expresión> siempre devuelve true.

Y la false de estas otras:

- false **and** <expresión> devuelve siempre false.
- false **or** <expresión> devuelve el valor de <expresión>.

7.2.3. Integer

Las instancias de la clase Integer representan números enteros. Los rangos de los enteros no están limitados. Dichas cotas dependen de las limitaciones en los recursos, el comportamiento cuando se sobrepasan dichas limitaciones es definido por la implementación.

7.2.4. Float

Las instancias de la clase Float representan números en coma flotante. La precisión depende de la implementación pero de forma estandar se usan 64 bits.

7.2.5. Array

Las instancias de la clase Array representan arrays. Además contendrán sólo elementos del mismo tipo, pertenecientes a las clases no compuestas, tales como integer, float, char o boolean.

7.2.5.1. Metodos de la clase Array

7.2.5.1.1. Operador new(tam, val)

Crea un nuevo objeto de la clase Array de tamaño val.

Por ejemplo, un array de tamaño 3 cuyos elementos se inicializan con el valor 7.5.

```
a = Array.new( 3, 7.5 )
```

O, por otro lado, también se pueden crear de la siguiente forma:

```
a = [1,2,3,…,n]
```

7.2.5.1.2. Operador []

Acepta como argumento un entero i. Devuelve el carácter que se encuentra en la posición i.

7.2.5.1.3. *[index] = arg*

Asigna a la posición index el valor de arg.

7.2.5.1.4. each var (bloque)

Ejecuta el bloque tantas veces como elementos tenga el array y asigna a var el contenido de la posición i correspondiente a la ejecución i del bloque

7.2.5.1.5. length

Devuelve el tamaño del array.

8. Sistema de ficheros

Todo el código, utilidades y pruebas del analizador sintáctico / semántico de Emerald se encuentra en la carpeta anexa *src*. El contenido de la misma se lista a continuación:

- **bisonear.sh** : utilidad para generar el analizador sintáctico / semántico a partir de las especificaciones léxica, sintáctica y semántica (Ver apartado 10.1).
- **lexicon.l :** especificación léxica.
- **sinta.x :** especificación sintáctica.
- **symbolsTable.h y symbolsTable.c :** contienen las funciones que manejan directamente la tabla de símbolos (inserción, búsqueda, etc).
- **semanticUtilities.h y semanticUtilities.c :** aglutinan las funciones propias del análisis semántico: chequeo y verificación de tipos, expresiones, llamadas a métodos, etc.
- **lexical_utilities :** utilidades que se emplearon para verificar el analizador léxico cuando éste era un programa independiente.
- **pruebas/ :** carpeta que contiene las pruebas para verificar el correcto funcionamiento de los analizadores léxico y semántico. En el apartado 12 se detalla la organización y contenido de esta carpeta.

9. Especificación léxica

Para la especificación léxica hemos recurrido a la utilidad FLEX (**F**ast **LEX**ical Analyzer), una herramienta libre que genera el analizador léxico correspondiente a una especificación léxica dada. La especificación léxica del lenguaje Emerald se encuentra en el fichero anexo *src/lexicon.l*.

9.1. Utilidades

Junto al fichero FLEX con la especificación léxica se incluye una carpeta *lexical_utilities* con utilidades para la generación del analizador léxico y la realización de las pruebas posteriores. A continuación se resumen las distintas herramientas proporcionadas:

9.1.1. Script *generar.sh*

Script para el intérprete de órdenes Bash de GNU/Linux que recibe un fichero FLEX como argumento (sin especificar la extensión). Esta utilidad realiza secuencialmente los dos pasos para obtener el analizador léxico a partir de la especificación léxica: generación del código C y la posterior compilación de este.

9.1.2. Programa *parser.rb*

Programa realizado en Ruby* que recibe un vector de identificadores numéricos (tal como sería una salida normal de un analizador léxico FLEX) y devuelve un nombre descriptivo para cada uno (por ejemplo, para el identificador 257 devolvería INTEGER).

(*) Se necesita el intérprete de Ruby para ejecutarlo.

9.1.3. Script *check.sh*

Script para el intérprete de órdenes Bash de GNU/Linux que recibe dos argumentos: un analizador léxico FLEX (el ejecutable) y un fichero con el código de pruebas. Esta utilidad ejecuta el analizador sobre el fichero de prueba y el resultado obtenido (la lista de tokens expresados como números) se le suministra a la utilidad *parser.rb* anterior, obteniendo así los nombres de los identificadores.

10. Especificación sintáctica y semántica (Bison)

Para la especificación sintáctica y semántica recurrimos a la pareja FLEX + Bison, siendo este último un generador de analizadores sintácticos dada una gramática. La gramática de Emerald se encuentra en el fichero anexo *src/sintax.y*, mientras que las funciones semánticas se guardan en los ficheros *symbolsTable* (.h y .c) y *semanticUtilities* (.h y .c).

10.1. Utilidad *bisonear.sh*

Junto al fichero Bison con la especificación sintáctica se incluye un script auxiliar denominado *bisonear.sh.* Este recibe como argumento la rutas de la especificaciones sintáctica y léxica y genera el analizador sintáctico correspondiente.

11. Entendiendo la tabla de símbolos (Ficheros symbolsTable.h/c)

En primer lugar hay que recordar que nuestro lenguaje Emerald desciende de un lenguaje interpretado donde no se declara el tipo de las variables. Esto nos lleva a la realización de **múltiples pasadas** durante el análisis para completar la tabla de símbolos, y nos fuerza a usar un **árbol de símbolos**, en lugar de una pila.

11.1.1. La estructura *Symbol*

La estructura principal y que supone el nodo de nuestro árbol de símbolos, es la estructura *Symbol*, la cual tiene la siguiente forma:

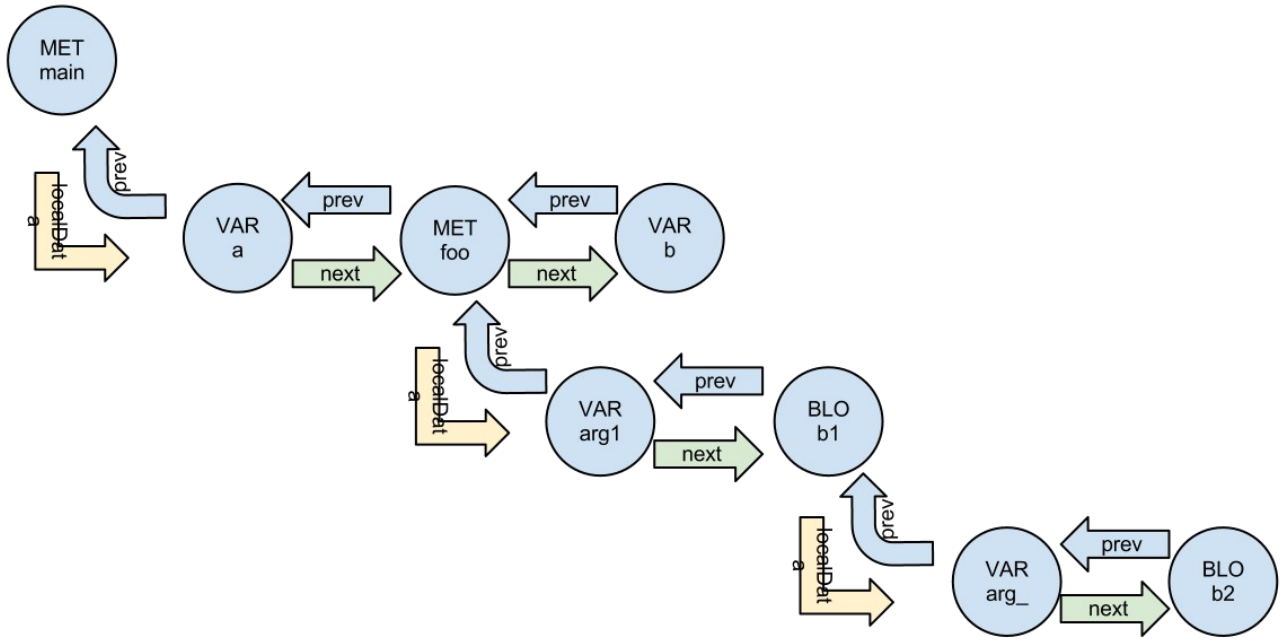
```
struct Symbol
{
    int symType;
    char *name;

    void *info;

    char firstChild;
    struct Symbol *prev, *next;
};
```

El significado de cada campo se detalla a continuación.

- **symType :** define el tipo de símbolo (tipo, variable, método, etc).
- **name :** nombre del símbolo
- **info :** puntero genérico que apunta al resto de información específica propia de cada tipo de símbolo. Por ejemplo, si *symType* fuera igual a *SYM_VARIABLE*, info apuntaría a una estructura de tipo *Variable* que contiene un puntero al tipo de la variable.
- **firstChild, prev, next :** el conjunto de símbolos locales a cada ámbito (función main, método o bloque) se implementa como una lista doblemente enlazada. Para cada símbolo, el puntero *next* siempre apunta al símbolo siguiente en el ámbito actual (hermano). El puntero *prev*, sin embargo, puede apuntar al hermano anterior, si *firstChild* = 0, o al padre, si firstChild = 1. Esto quizás se vea mejor con un diagrama.



Ejemplo de árbol semántico. VAR = Variable, MET = Method, BLO = Block

En la imagen se muestra un ejemplo de árbol semántico sencillo. Se observa cómo el puntero *next* apunta siempre al siguiente hermano en el mismo nivel, mientras que *prev* puede apuntar al hermano anterior o al padre. En este ejemplo, las variables arg1 y arg_ tendrán su campo *firstChild* a 1 para indicar que mediante *prev* apuntan al padre.

11.1.2. Variables globales

El funcionamiento de la tabla de símbolos se apoya en el uso de una serie de variables globales:

- **char nextSymIsFirstChild :** cuando se inserta un método, esta variable se pone automáticamente a uno para indicar que el siguiente símbolo que se introduzca en la tabla será el primer hijo (o dato local) de dicho método (al menos que se especifique lo contrario).

- **Method* lastDefinedMethod :** último método / bloque definido. Útil a la hora de insertar variables en el contexto actual.

- **Symbol* mainMethodNext :** puntero que apunta al primer hijo (símbolo local) de la función main. Es útil cuando se desea buscar un método, ya que como no permitimos submétodos, todos los métodos son hermanos de este símbolo.

- **Symbol* mainMethod :** puntero a la raíz del árbol semántico.

- **char change :** variable “booleana” que indica si el árbol de símbolos se ha modificado o no en la pasada actual.

12. Ficheros de prueba

Anexa a esta memoria se incluye una carpeta *src/pruebas/* con dos subcarpetas *lexico/* y *sintactico/*. La primera contiene una serie de ficheros consistentes en secuencias de tokens para verificar el correcto funcionamiento del analizador léxico. La carpeta *pruebas/sintactico* contiene programas escritos en Emerald para testear el correcto funcionamiento del analizador *sintáctico/semántico*.

12.1. Pruebas del analizador léxico

12.1.1. Fichero *identificadores.em*

Sucesión de identificadores de variables globales, locales y constantes.

12.1.2. Fichero *literales_numericos.em*

Secuencia de literales numéricos (enteros y números en coma flotante).

12.1.3. Fichero *literales_string.em*

Secuencia de literales string (incluye caracteres).

12.1.4. Fichero *palabras_clave.em*

Secuencia de palabras clave escritas correctamente e incorrectamente.

12.2. Pruebas del analizador sintáctico / semántico

12.2.1. Fichero *asignaciones.em*

Secuencia de asignaciones a distintos tipos de variables.

12.2.2. Fichero *clase.em*

Ejemplo sencillo de definición de una clase y acceso a sus campos.

12.2.3. Fichero *es_if_anidados.em*

Prueba de E/S más una sucesión de estructuras condicionales if anidadas a dos niveles.

12.2.4. Fichero *es_if_.em*

Prueba de E/S más una estructura condicional if.

12.2.5. Fichero *llamada_each.em*

Programa que recorre un array imprimiendo sus elementos por pantalla.

12.2.6. Fichero *llamadas_funciones_anidadadas.em*

Multiplicación de los elementos de un vector por una constante. Prueba de una llamada a método dentro de un bloque (método anónimo).

12.2.7. Fichero *minicalc.em*

Intérprete interactivo de expresiones aritméticas enteras.

12.2.8. Fichero *precedencia_operadores.em*

Ejemplo de expresión con distintos operadores.

12.2.9. Fichero *tratar_string.em*

Aplicación que toma una string y va sustituyendo cada carácter por el número de su posición (se le suma el código ASCII del carácter '0' para que se muestre bien).

13. Bibliografía

13.1. Especificación informal

Programming Languages — Ruby . IPA Ruby Standardization WG Draft . Information-technology Promotion Agency, Japan 2010

Ruby Loops - while, for, until:

http://www.tutorialspoint.com/ruby/ruby_loops.htm

Ruby Programming/Syntax/Literals:

http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Literals

The Ruby Language:

http://www.ruby-doc.org/docs/ProgrammingRuby/html/language.html

Ruby.new:

http://www.ruby-doc.org/docs/ProgrammingRuby/html/intro.html

Ruby Operators:

http://www.tutorialspoint.com/ruby/ruby_operators.htm

The Ruby Case Statement:

http://www.techotopia.com/index.php/The_Ruby_case_Statement

Understanding Ruby Blocks, Procs and Lambdas:

http://www.robertsosinski.com/2008/12/21/understanding-ruby-blocks-procs-and-lambdas/

Ruby Blocks 101:

http://allaboutruby.wordpress.com/2006/01/20/ruby-blocks-101/

Documentación de Ruby:

http://www.ruby-doc.org/core-1.8.7/

13.2. Especificación léxica

Lexical Analysis With Flex, for Flex 2.5.37:

http://flex.sourceforge.net/manual/

ANEXO – Lista de cambios con respecto a las anteriores entregas

Memoria

- Hasta ahora se habían realizado entregas parciales. Esta memoria es la primera que unifica y amplia a todas las anteriores.
- Corrección de erratas.

Léxico

- Añadidos los tokens EACH, NEW y ARRAY.
- Se ha modificado el token ID_CONSTANT. Antes se forzaba que todas las letras del identificador fueran mayúsculas. Ahora sólo se fuerza que lo sea el primer caracter.

Sintáctico

- Se ha añadido tratamiento de errores (derivación con el token "error" y mensaje de error) en gran parte de los símbolos no terminales.
- No terminal *method_call*
  - Añadida la posibilidad de invocar un método sin argumentos empleando únicamente su identificador (sin paréntesis vacíos).
- No terminal *arguments*
  - Añadida la posibilidad de expresar la lista de argumentos entre paréntesis.
- No terminal *method_call_argument*
  - Ahora se permite que un argumento sea una expresión o una string.
- No terminal *more_arguments*
  - Solucionado un error por el que se entraba en un bucle infinito.
- No terminal *block_call*
  - Solucionado un error por el que permitíamos la llamada a un bloque con más de un argumento (en Emerald sólo permitimos pasar bloques de un sólo argumento al método "each").
- No terminal assignment"
  - Se ha simplificado su definición mediante la inclusión de los no terminales auxiliares

"left_side", "attribute" y "right_side".

- Se han añadido los no terminales content_vector, code, method_code y sentences.
  - Añadida sintaxis de strings compuestas.