Lenguaje de programación Emerald

Procesadores de Lenguajes

Aridane J. Sarrionandia de León
Garoe Dorta Pérez
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.

2. Conceptos fundamentales

2.1. Variables

Una variable es una asociación entre un identificador y su valor asociado.

2.1.1. Ámbito de las variables

En Emerald diferenciamos entre cuatro tipos de ámbitos para las variables: **constantes**, **locales**, **globales** y **de instancia**.

2.1.2. Tipos de variables

Las **variables simples** se clasifican en **enteros**, **floats**, **caracteres** y **boleanos**.

Por su parte, las **variables compuestas** incluye los **arrays** y las **clases**. Tanto los unos como las otras sólo pueden contener variables simples.

2.2. Métodos

Procedimientos / funciones del lenguaje. Únicamente aceptan argumentos de entrada simples (ni arrays ni estructuras) y pueden devolver como máximo un único valor de retorno simple.

2.3. 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 }
```

2.4. 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

Una instancia de una clase es un **objeto**.

NOTA: las clases no están implementadas a nivel de código, pero si a nivel semántico.

2.5. 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.

2.6. Recursividad

Emerald admite recursividad.

2.7. Entrada/Salida

La salida estándar en Emerald sigue la forma:

puts(<string>)

Donde <string> puede contener variables en su interior (esto se explicará más adelante).

Por su parte, la entrada sigue el formato:

<variable> = getc()

La instrucción anterior hace que se guarde en "<variable>" el **caracter** introducido por el usuario, para enteros y números en coma flotante se emplearía geti y getf, respectivamente.

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, tabuladores, comentarios, marca fin de programa o tokens.

3.1. Comentarios

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

a + b # Esto es un comentario.

3.2. Tokens

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

3.2.1. 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	puts
getc	gets	getf			

Nota: las palabras clave son case-sensitive.

3.2.2. 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_])*

3.2.3. Signos de puntuación (punctuators)

Los signos de puntuación admitidos son los siguientes:

[	]	{	}
(	)	,	;

3.2.4. Operadores

Lista de operadores de Emerald:

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

Los @ de los operadores + y – hacen referencia a que son operadores unarios. Pero esto no quiere decir que se usen con el @.

3.2.5. Literales

En Emerald diferenciamos entre literales numéricos, strings y arrays.

3.2.5.1. *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:

- - 123: hay un espacio en medio, - es el operador binario.
- -123: no hay espacios en medio, - es el operador unario.

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

3.2.5.2. 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.

3.2.5.3. 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.

3.2.5.4. *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 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)

   z = x + y

end

Suma( a, b)
```

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

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 entre paréntesis.

Por ejemplo:

suma(1, 2)

[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 entre paréntesis. <cuerpo_método> es el código interno del método.

Nota: dentro de <cuerpo_método> no se permite definir clases ni definir un método interno.

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 bloque:

```
array_var = [1, 2, 3, 4]
b = 1
array_var.each do |n|
    i = n
    b = n
    puts(i)
end
a = i
a = n
puts(b)
```

Tanto **a = i** como **a = n** darán un error al compilar puesto que las variables **i** y **n** solo son accesibles desde dentro del bloque. Sin embargo, la variable **b** al ser externa al bloque contendrá el último valor del array. El método each de la clase Array recorre el array y para cada elemento ejecuta el bloque suministrado como parámetro, por lo que en este ejemplo se mostraría el contenido del array. En conclusión, un bloque se comporta como un método pero tiene acceso a la variables locales que se han declarado en su ámbito.

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

```
array.each { |n| puts(n) }
```

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:

- Declaración de un array (Array.new o [a_1, a_2, …, a_n])
- Una expresión, la cual puede incluir literales, variables simples y/o elementos de array.

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, números en coma flotante y caracteres.

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. Expresión condicional if

Sigue la forma:

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

<sentencias>

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

end

6.5.1.2. 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.

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 pueden ser variables 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. Por ejemplo:

```
p = Punto.new
```

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 que implementa Emerald es de 32 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 iteración i-ésima del bloque

8. Sistema de ficheros

Todo el código, utilidades y pruebas del compilador de Emerald se encuentra en la carpeta anexa *src*. El contenido de la misma se resume a continuación (cada uno se desarrollará en apartados posteriores):

- **lexicon.l :** especificación léxica.
- **lexical_utilities :** utilidades que se emplearon para verificar el analizador léxico cuando éste era un programa independiente.
- **sinta.x :** especificación sintáctica / semántica.
- **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.
- **codeGenUtils.h y codeGenUtils.c :** contienen las funciones encargadas de generar el código final (cuádruplas en máquina Q).
- **Q.h, Qlib.h y Qlib.c :** ficheros de la máquina Q.
- **bisonear.sh, quar.sh** : utilidades empleadas para la generación del analizador y la realización de pruebas.
- **tests/ :** carpeta que contiene las baterías de pruebas divididas en tres subcarpetas (*lexicon*, *semantics* y *code*) según el ámbito de las mismas.

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 distintas utilidades para la generación del analizador léxico y la realización de las pruebas posteriores.

Nota: las utilidades mencionadas dejaron de usarse cuando el analizador léxico dejó de ser un programa independiente, mientras que el analizador como parte del resto del compilador ha sufrido modificaciones.

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.

9.2. Pruebas léxicas

La carpeta *src/tests/lexicon/* contiene una serie de ficheros con identificadores, palabras clave y literales de Emerald para verificar el correcto funcionamiento del analizador léxico.

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. 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.

10.1.1. 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:

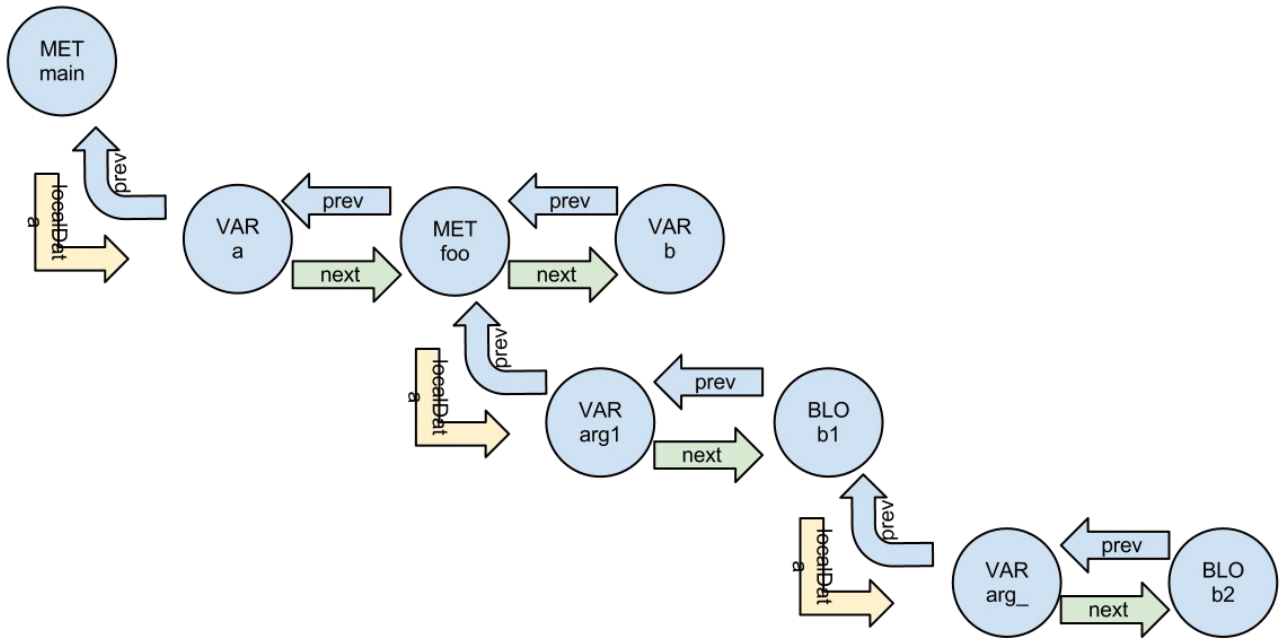
```
struct Symbol {
    int symType;
    char *name;

    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. En ambos casos, el padre apunta al hijo mediante un puntero *localData* situado en su estructura auxiliar *info* de tipo Method.

10.1.2. Estructuras auxiliares info

Como se ha nombrado anteriormente, el campo info de la estructura Symbol apunta a un tipo de estructura o a otro según el tipo de símbolo que estemos tratando. A continuación se detallan las diferentes estructuras posibles.

10.1.2.1. Estructura Type

Esta estructura es usada en los símbolos de tipo type, que indican los tipos de las variables. Los campos son los siguientes:

- **int id:** identificador numérico único para cada tipo.
- **unsigned int size** (generación de código)**:** tamaño en bytes del tipo.
- **struct ArrayType* arrayInfo** y **struct ClassType* classInfo:** estructuras auxiliares metidas en un **union** de C ya que nunca se usarán ambas al mismo tiempo. Contienen información extra si el tipo que estamos tratando es un array o una clase, respectivamente.

10.1.2.1.1. Estructura ArrayType

Guarda información sobre un tipo array. Sus campos son los siguientes:

- **Symbol* type:** puntero al tipo de los elementos del array.
- **unsigned int nElements:** número de elementos del array.

10.1.2.1.2. Estructura ClassType

Contiene la información de un tipo clase. Sus campos son los siguientes:

- **unsigned int nElements:** número de campos de la clase.
- **struct Symbol **elements:** vector de punteros a los campos de la clase.

10.1.2.2. Estructura Variable

Se refiere a símbolos de tipo variable. Esta estructura contiene los siguientes campos:

- **Symbol* type:** puntero al tipo de la variable.
- **int symSubtype** (generación de código)**:** subtipo de la variable (local, argumento).
- **int address** (generación de código)**:** dirección de la variable en memoria (generación de código). En el caso de variables globales, address indica la dirección exacta en memoria, mientras que en las variables locales y los argumentos indica un desplazamiento con respecto a R6.

10.1.2.3. Estructura Method

Guarda información sobre un método o un bloque. Sus campos son los siguientes:

- **int nArguments:** número de argumentos del método o bloque.
- **Symbol *lastSymbol:** puntero al último símbolo local del método o bloque.
- **Symbol *localSymbols:** puntero al primer símbolo local del método o bloque.
- **Symbol *returnType:** puntero al tipo de la variable de retorno.
- **int argumentsSize, localsSize** (generación de código): tamaño total en bytes de los argumentos y de las variables locales del método / bloque, respectivamente.
- **int label** (generación de código): etiqueta que se asociará al método / bloque en el código final.

10.1.2.4. Estructura auxiliar ExtraInfo

Los símbolos con esta estructura en su campo info son estructuras temporales (no se guardan en la tabla de símbolos). Se emplean para casos como las asignaciones, donde se necesita información extra además de los símbolos implicados. Es utilizada exclusivamente durante la generación de código. Sus campos se listan a continuación:

- **int assignmentType** (generación de código)**:** diferencia si se trata de una asignación para una variable simple o para un array / clase.
- **int nRegister** (generación de código)**:** registro resultado de la expresión de la parte derecha de la asignación.
- **struct Symbol* variable** (generación de código)**:** símbolo asociado a la variable de la parte izquierda de la asignación.

10.2. Estructura auxiliar SymbolInfo

Esta estructura es usada en las reglas de la gramática cuando es necesario no solo devolver un tipo struct Symbol sino también información extra. Tiene los siguientes campos:

- **Symbol *symbol:** puntero al struct symbol que se quiere devolver.
- **Symbol *varSymbol** (generación de código)**:** apunta al símbolo de la variable.
- **Symbol *exprSymbol** (generación de código)**:** apunta al extraInfo asociado a expresión0 en asignaciones del tipo array[expresión0] = expresión1.
- **int info:** información extra que se interpreta según el contexto.
- **char *name:** nombre del campo de la clase cuando symbol apunta a una variable de clase.

- **int nRegister** (generación de código): registro con el resultado de la expresión que se asigna.

10.3. 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* mainMethod :** puntero a la raíz del árbol semántico.
- **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.
- **char change :** variable “booleana” que indica si el árbol de símbolos se ha modificado o no en la pasada actual.

10.4. Ejemplo de tabla de símbolos

Para resaltar algunas características de la tabla de símbolos presentamos el siguiente ejemplo:

```
def fibonacci( n )
   if ( n >= 2 )
      n = fibonacci( n - 1 ) + fibonacci( n - 2 )
   end
   n = n
end

puts ( "Input the value to calculate fibonacci's number: " )
n = geti
i = fibonacci( n )
puts ( "Fibonacci's number for #{n} is #{i}" )
```

Produce como resultado la siguiente tabla de símbolos, (cada tabulador índica que nos hemos metido por el campo localSymbols – entrado en un ámbito -):

METHOD - [_main]- arguments' size [0]- locals' size [8]- label [0] - return type: [NULL]]- label [0] - nArguments: [0] - hijo: [puts]

METHOD - [puts]- arguments' size [8]- locals' size [0]- label [1] - return type: [NULL]]- label [1] - nArguments: [1] - hijo: [input_str]

VARIABLE - [input_str] - type: [array_char_0] - id: [7] - address [8]

METHOD - [getc]- arguments' size [8]- locals' size [0]- label [2] - return type: [char]- label [2] - nArguments: [0] - hijo: [NULL]

METHOD - [exit]- arguments' size [8]- locals' size [0]- label [3] - return type: [NULL]]- label [3] - nArguments: [0] - hijo: [NULL]

TYPE - [integer] - id:[1] - size:[4]

TYPE - [float] - id:[2] - size:[4]

TYPE - [char] - id:[3] - size:[1]

TYPE - [boolean] - id:[5] - size:[1]

TYPE - [array_char_0] - id:[7] - size:[0] - type:[char] - nElements:[0]

METHOD - [fibonacci]- arguments' size [12]- locals' size [0]- label [4] - return type: [integer]- label [4] - nArguments: [1] - hijo: [n]

VARIABLE - [n] - type: [integer] - id: [1] - address [8]

VARIABLE - [n] - type: [integer] - id: [1] - address [4]

VARIABLE - [i] - type: [integer] - id: [1] - address [8]

Hay varios puntos a tener en cuenta:

- Se inserta de manera automática los siguientes símbolos:
  - Método main: puntero al método main, que es la raíz del árbol.
  - Metodos puts y getc: métodos de entrada/salida proporcionados por el lenguaje.
  - Tipos integer, float, char, boolean y array_char_0 (string): tipos básicos para las variables.
- Los argumentos de los métodos y bloques se crean como variables locales en la tabla de símbolos.

10.5. 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.

10.6. Pruebas semánticas

La carpeta anexa src/tests/semantics incluye diferentes ficheros de prueba para verificar el correcto funcionamiento del analizado sintáctico / semántico.

11. Generación de código

La generación de código consta de dos partes diferencias:

- La **generación del "código de usuario"** (ficheros *codeGenUtils.h* y *codeGenUtils.c*): conjunto de funciones que generan los fuentes *.q.c a partir del código fuente escrito en Emerald.

- Las "**llamadas al sistema**" incorporadas a la máquina Q (ficheros *Qlib.h* y *Qlib.c*): funciones de E/S que se invocan desde el código de usuario.

11.1. Generación del código de usuario

Se trata del conjunto de funciones que traduce los fuentes escritos en Emerald en fuentes de la máquina Q. Se encuentran en los ficheros *codeGenUtils.h* y *codeGenUtils.c* y se diferencian varios grupos de funciones:

- **Gestión de registros libres / ocupados:** se trata de funciones para reservar registros y liberarlos cuando no se necesiten más, así como el derramado y el salvado de los mismos (funciones *assignRegisters*, *freeRegister* y *freeRegisters*, etc).

- **Generación efectiva de código:** conjunto de funciones de la forma gen* (pe. *genMethodBegin*, *genMethodCall*) generan bloques de código de usuario según varios parámetros suministrados como argumentos.

- **Funciones auxiliares:** funciones pequeñas de uso común (pe. *newLabel* para generar una nueva etiqueta).

11.2. Llamadas al sistema

Los ficheros *Qlib.h* y *Qlib.c* se modificaron para incluir la llamada **puts** (imprimir una string) y la **familia de llamadas get** (geti, getf, getc) para leer la entrada del usuario.

11.3. La utilidad *quar.sh*

Para acelerar la realización de las pruebas se generó un script *quar.sh* (en la carpeta *src*). Su uso es el siguiente:

./quar.sh <*test.em*>

Donde *test.em* es un programa cualquiera escrito en Emerald. El script realiza entonces los siguientes pasos:

1. Llama a "*./bisonear.sh sintax.y lexicon.l*" para generar el compilador.
2. Ejecuta el compilador anterior sobre el fichero de pruebas (*test.em*). Este paso genera (haya o no haya error) un fichero *out* con información de depuración.
3. Compila la máquina Q (por si se hubiera realizado algún cambio en su código).
4. Ejecuta la máquina Q con el fichero *test.q.c* generado por el compilador.

Si se produce algún error en los pasos del 1 al 3, el script aborta la ejecución.

11.4. Pruebas de código

La carpeta anexa src/tests/*code* incluye diferentes ficheros de prueba para verificar el correcto funcionamiento del analizado sintáctico / semántico.

12. Bibliografía

12.1. 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/

12.2. Especificación léxica

Lexical Analysis With Flex, for Flex 2.5.37:

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