Compilador de LOGO a HTML5

César Enrique Ramírez

Junio 2019

Abstract

Implementacion de un compilador del lenguaje logo para representarlo en un navegador usando un canvas. La implementacion está realizada con el lenguaje de programacion Scala

1 Introducción

El objetivo del trabajo es analizar un código escrito en lenguaje LOGO y transformarlo en código HTML5 utilizando un canvas y JavaScript. Se nos entrega una descripción del lenguaje en forma de gramática EBNF, que tras un primer análisis vemos que no es LL(1).

Para facilitar la implementación de un compilador para dicha gramática haremos uso de una herramienta que previamente era de la librería estandar de Scala, pero que ahora mismo es un modulo externo: Parser Combinators

2 Instrucciones de uso

2.1 Requerimientos

El proyecto utiliza Scala como lenguaje y SBT como build tool. No debería ser necesario tener instalado nada previamente, puesto que la distribución de SBT que lleva el proyecto debería ser suficiente para descargar todo lo demás.

2.2 Cómo compilo el código

Para compilar el proyecto hay que ejecutar el comando:

```
sbt compile
```

Para generar un jar ejecutable:

```
sbt assembly
```

Dicho jar se encontrará en el directorio:

```
target/scala-SCALA_VERSION/logo-compiler-assembly-VERSION.jar
```

2.3 Cómo ejecuto el jar

En el directorio bin/ se encuentra un script ejecutable que facilita el proceso. Simplemente hay que llamar a dicho script pasándole como parámetro la ruta al archivo .logo que queramos compilar. El resultado estará en un archivo out.html en el directorio de trabajo

3 Descripción del compilador

Para la implementación de este compilador hemos establecido 4 etapas independientes:

1. Análisis léxico
2. Análisis sintáctico
3. Generación de código y análisis semántico
4. Insertado de código en una plantilla html

Respecto a las decisiones de diseño que se han tomado, podemos destacar:

1. Los parámetros que reciben los procedimientos y los bucles se implementan como una variable más que se define en el momento de ejecutarse el bloque correspondiente
2. Los procedimientos son los únicos que tienen la funcionalidad de ámbitos con respecto a las variables, es decir, que las variables que se definen dentro de un procedimiento no afectan a el exterior
3. Pero tampoco tienen visibilidad de las variables de fuera. Solo de aquellas que se les pasa por parámetro
4. La implementación soporta funciones recursivas
5. Las funciones "nativas" implementadas son: random, sin, cos, mod

3.1 Análisis léxico

El análisis léxico lo lleva a cabo LogoLexer que se encuentra en el paquete parser. LogoLexer no es más que un objeto que extiende de RegexParsers, una "interfaz/clase" del módulo que mencionamos la princio Parser Combinators que nos proporciona ciertas características para describir un Lexer basado en expresiones regulares. Así, cada uno de los tokens que podemos identificar, estará descrito por una expresión regular.

Son dichas expresiones regulares las que se encargan de distinguir los caracteres que consideramos blancos, y comentarios, mediante un campo especial:

```
override val whiteSpace = "(\\s|;.*)+".r
```

La posición de cada token en el flujo de entrada original queda registrada al ser parseado dicho token. Esto es posible gracias a que la clase LogoToken extiende de la interfaz Positional que proporciona el campo pos. Dicho campo lo rellena el método positioned en el momento de parsear el token correspondiente.

El análisis léxico tiene como resultado una lista de tokens, pero esto va "envuelto" en un tipo de dato que nos permite representar posibles errores:

```
Either[LogoLexerError, List[LogoToken]]
```

El tipo Either representa un tipo producto, es decir, que el resultado puede ser de tipo LogoCompilationError o de tipo List[LogoToken]. Para poder acceder al resultado, tenemos que hacerlo mediante ciertos métodos que interactuan con el tipo Either con ciertas características comunes con lo que podríamos llamar Monad. Aún así, no es necesario conocimiento alguno sobre el tema.

3.2 Análisis sintactico

El análisis sintáctico lo lleva a cabo LogoParser que se encuentra en el paquete parser. LogoParser es un objeto cuyo método principal recibe una lista de tokens y devuelve un objeto de tipo LogoAST.

El tipo LogoAST está definido en el archivo LogoAST dentro del paquete parser y no es más que una estructura de traits y case classes mediante la cual representamos la estructura lógica del lenguaje

La implementación del análisis sintáctico es una traducción casi directa de las producciónes de la gramática al DSL (Domain Specific Language) que nos facilita el módulo Parser Combinators.

Al igual que LogoLexer, LogoParser también genera su resultado "envuelto" en el tipo Either:

```
Either[LogoParserError, LogoAST]
```

3.3 Generación de código y análisis semántico

La parte más compleja es, sin duda, el evaluador y generador de código. La tarea de la evaluación de expresiones la lleva a cabo ExpresionEvaluator, mientras que de la evaluación del resto del ast y de la generación de código se encarga LogoCodeGenerator, ambos localizados en el paquete evaluator.

El proceso de evaluación del AST lo podemos resumir en los siguientes pasos:

1. registrar los procesos en la tabla de simbolos
2. evaluar una a una todas las instrucciónes del programa

A la hora de evaluar las instrucciones de forma sequencial, nos dotamos de unas estructuras de datos que nos permiten mantener el sistema libre de variables mutables, es decir, que todo el proceso de evaluación es funcional.

Para llevar esto a cabo, utilizamos un fold que va acumulando y combinando paso a paso el código generado y los cambios en la tabla de simbolos y en las variables internas. Para explicar un poco más detalladamente el proceso, tengo que introducir una descripción básica de dos tipos de datos utilizados: SymbolTable y EvalData

3.3.1 SymbolTable

Se utiliza para registrar las variables, los procedimientos y valores internos como la posición de la tortuga o el estado del lápiz.

- Las variables están representadas como un campo de tipo Map[String, Int], por lo que obligatoriamente todas las variables tienen que ser de tipo entero
- Los procedimientos están representados como un campo de tipo Map[String, Procedimiento]. A la hora de evaluar una llamada a un procedimiento, se comprueba que esté definido aquí, y en caso afirmativo, se procede a evaluar el objeto Procedimiento correspodiente, utilizando las variables actuales
- El resto de valores internos están definidos como campos de tipos sencillos

En el proceso de evaluar una instrucción tras otra, cada evaluador va devolviendo (dentro de EvalData, del que hablaremos a continuación) una instancia de SymbolTable con todos los valores que tenía cuando él la recibió, más todas las variables y/o procedimientos que se hallan definido en el camino.

3.3.2 EvalData

EvalData representa toda la información que le va pasando un evaluador al siguiente durante el proceso de evaluación de las intrucciones. Contine información sobre posibles errores que hallan sucedido, el código que se ha ido generando, y la tabla de símbolos. Cabe destacar que todo el compilador sigue una táctica de gestión de errores fail fast, es decir, que si se produce un error, solo se devolverá ese error, puesto que todo lo demás será irrelevante (más adelante hablaré un poco más sobre la gestión de errores).

Para hacer posible que toda esta estuctura funciones, es necesario que combinemos el código que se ha ido generando (y que contiene el objeto EvalData) mientras que solo nos vamos quedando con la tabla de símbolos que devolvión la última evaluación. Esto es posible gracias a que EvalData es un Monoid. Un monoid se define como una categoría de cosas que tienen un operador de combinación y uno de elemento neutro. La combinación de EvalData concatena las instrucciones de codigo generado, mientras que se queda con la tabla de símbolos del objeto EvalData de la derecha.

Monoid está definido en el paquete util de la siguiente manera:

```
1 trait Semigroup[A] {
2     def combine(x: A, y: A): A
3 }
4
5 trait Monoid[A] extends Semigroup[A] {
6     def empty: A
7 }
```

3.3.3 ExpresionEvaluator

La evaluación de expresiones la realiza ExpresionEvaluator que se encuentra en el paquete evaluator. Esta tarea es compleja por sí misma, así que consideramos

que necesitaba ser aislada del resto para poder tratarla adecuadamente.

El evaluador de expresiones recibe las expresiones a evaluar, junto con una tabla de símbolos, que usa para poder saber el valor de variables. La funcionalidad por defecto supone devolver un objeto de tipo ValueResult que codifica la posibilidad de que el resultado de la evalución sea un error:

```
type ValueResult = Either[LogoEvaluationError, Int]
```

Pero el evaluador también proporciona algunos métodos útiles, donde en vez de recibir una expresion y una tabla de simbolos, y devolver un ValueResult, recibe 1, 2, 3, o n expresiones, una tabla de símbolos, y una función de enteros a EvalData, permitiendo así el fácil acceso a los valores que representan las expresiones, pero sin sacrificar la gestión de los errores.

```
1 def evalN(expresiones: List[Expresion], s: SymbolTable)
2         (f: List[Int] => EvalData)
3 : EvalData = {
4   import util.EitherUtils._
5   expresiones.map(ExpresionEvaluator(_, s)).sequence match {
6     case Left(err) => EvalData(Left(err), s)
7     case Right(values) => f(values)
8   }
9 }
```

```
1 def eval2(expr1: Expresion, expr2: Expresion, s: SymbolTable)
2         (f: (Int, Int) => EvalData)
3 : EvalData = {
4   val ff: List[Int] => (Int, Int) = { case a :: b :: _ => (a, b) }
5   evalN(List(expr1, expr2), s)(f.tupled.compose(ff))
6 }
```

3.4 Gestion de Errores

Para la gestión de los errores es para lo que utilizamos algo que venimos nombrando desde el comienzo: Either. El tipo Either nos permite representar en el tipo de retorno la posibilidad de fallo, haciendo así que nuestro programa esté libre de excepciones (al menos intencionadas) en la gestión de los errores. La funcionalidad que nos permite esto es que Either es un tipo que se puede "componer" con otros objetos de su mismo tipo, es decir, que dados dos objetos de tipo Either podemos generar un tercero, que representa la mezcla de los dos anteriores: si combinamos un Either que contiene un error con cualquier cosa, el resultado será un Either con dicho error, si ambos contienen otra cosa, la combinación se realiza de acuerdo a como nosotros la definamos.[^Para más información en este tema, buscar sobre: Monads, flatMap, cláusula for en Scala.]

3.5 Ajustes en la plantilla html

Algunos ajustes han sido necesarios en la plantilla de html para poder generar el código adecuadamente. Estos ajustes son:

- Cambiar la orientación de la coordenada y del canvas
- Cambiar el centro de coordenadas del canvas

3.6 Caracteristicas que faltan

La implementación actual no incluye las siguientes características:

- Librerías: no están soportadas ni las instrucciones import ni la funcionalidad de buscar dicho código en otro archivo .logo
- ajuste automático del tamaño del canvas
- el Comando stop es reconocido, pero no hace nada
- al igual que los comandos hideturtle y showturtle

4 Código

El código puedes descargarlo de GitHub