Trabajo práctico final - ALP

Antinori Nicolás
A-3643/9

Resumen

Implementación de un compildor de un DSL derivado Berkeley Logo 6.0 (bautizado miniLogo por mi). El lenguaje Logo es un lenguaje dedicado para enseñar programación, puede usarse para enseñar la mayoría de los principales conceptos de la programación, ya que proporciona soporte para manejo de listas, archivos y entrada/salida, etc.
Una característica más explotada de Logo es poder producir **gráficos tortuga**, es decir, poder en dar instrucciones a una tortuga virtual, un cursor gráfico usado para crear dibujos, Esta tortuga o cursor se maneja mediante palabras que representan instrucciones. En esta versión del lenguaje, nos concentramos en la producción de dichos gráficos.

Índice

1. El lenguaje.

1.1. Estructura del código de fuente.

La estructura del código de fuente de un archivo de miniLogo constara de dos comandos de inicialización del documento, los cuales son:

- canvasSize n m: Establece las dimensiones de la imagen de salida.
- backColor ⟨*color*⟩: Establece como color de fondo de la imagen compilada el ⟨*color*⟩ elegido.

Estos comandos son **obligatorios**, deben estar al inicio del archivo y se debe respetar el orden.

1.2. Expresiones aritméticas.

1.2.1. Operaciones binarias.

- + : Suma convencional de numeros.
- - : Resta convencional de numeros.
- * : Producto convencional de numeros
- / : Cociente convencional de numeros

Nota: Los operadores antes mencionados son todos **infijos**.

1.2.2. Operaciones Unarias.

- - n : Devuelve el opuesto de n.
- MINUS n: Devuelve el opuesto de n.
- INT n: Devuelve la parte entera de n
- ROUND n: Redondea n.
- SQRT n: Nos devuelve la raiz cuadrada de n.
- EXP n: Nos devuelve el número e^n.
- LN n: Nos devuelve el logaritmo nataral de n.
- LOG10 n: Nos devuelve el logaritmo en base 10 de n.
- SIN n: Nos devuelve el seno de n, donde n esta en grados.
- RADSIN n: Nos devuelve el seno de n, donde n esta en radianes.
- COS n: Nos devuelve el coseno de n, donde n esta en grados.
- RADCOS n: Nos devuelve el coseno de n, donde n esta en radianes.
- ARCTAN n: Nos devuelve el arcotangente de n, donde n esta en grados.
- RADARCTAN n: Nos devuelve el arcotangente de n, donde n esta en radianes.

Nota: n puede ser tanto una constante numérica, como una variable.

1.2.3. Predicados numéricos.

- **LESSP**:

  LESSP n m
  LESS? n m
  n < m

  Nos devuelve **true** si n es estricamente menor a m.

- **GRATERP**:

  GRATERP n m
  GREATER? n m
  n > m

  Nos devuelve **true** si n es estricamente mayor a m.

- **LESSEQUALP**:

  LESSEQUALP n m
  LESSEQUAL? n m
  n `<=` m

  Nos devuelve **true** si n es menor o igual a m.

- **GRATEREQUALP**:

  GRATEREQUALP n m
  GREATEREQUAL? n m
  n `>=` m

  Nos devuelve **true** si n es mayor o igual a m.

- **EQUALP**:

  EQUALP n m
  EQUAL? n m
  n = m

  Nos devuelve **true** si n es igual a m.

1.3. Operaciones lógicas

- **AND** b_1 b_2: Nos devuelve **true** si tanto b_1 como b_2 son **true**, y **false** en otros casos.
- **OR** b_1 b_2: Nos devuelve **true** si b_1 es **true** o b_2 **true**, y **false** en otros casos.
- **NOT** b: Nod devuelve **true** si b equivale a **false** y viceversa.

Nota: b_i corresponde a una expresión booleana.

1.4. Definición de variables

Antes de ver como se declaran los distintos tipos de variables, debemos dejar en claro como hacer referencia a una palabra literalmente. Para esto, solo debemos añadir una comilla (“) delante de la misma. por ejemplo, la expresión `"hola` hace referencia a la cadena de texto “hola”.

- **MAKE**

  MAKE varname expr

  Asigna el valor de evaluar la expresión “expr” a la variable con el nombre “varname”. La expresión puede ser tanto booleana como numérica, “varname” debe ser una palabra literal. Los nombres de variables no son sensibles a cambios mayúsculas y minúsculas y si existe una variable con dicho nombre, la misma sera actualizada al valor de la nueva expresión.

- **LOCALMAKE**

  LOCALMAKE varname expr

  Idem anterior, con la diferencia de que se usa dentro de la definición de procedimientos, y la variable creada solamente sera vista dentro del mismo.

Para referirnos al valor de una variable ya creada, podemos hacer referencia a la misma simplemente nombrandola (sin las comillas (“) delante del nombre de la variable) o escribiendo dos puntos (:) delante del nombre. Por ejemplo:

```
MAKE "x 1
MAKE "y :x + 1
```

Lo que hace dicho código es, crear una variable de nombre “x”, asignarle el valor 1, y luego crear una variable “y” la cual contendrá el valor de la variable x, sumandole 1 unidad.

1.5. Definición de procedimientos

Los procedimientos se definen con la primitiva **TO** de la siguiente forma:

```
TO procname :x1 :x2 ... :xn [:y1 e1] [:y2 e2] ... [:ym em]
comandos
END
```

De esta forma se definirá un procedimiento con el nombre “procname” (el cual no debe ser una palabra literal), que estará preparado para recibir los argumentos **obligatorios** x_1, ...,x_n y los argumentos **opcionales** $y1$, ..., y_n. No debe existir otro procedimiento con el nombre “procname” y los nombres de los mismos no son sensibles a cambios mayúsculas y minúsculas.

Al llamar a un procedimiento, se deberá llamar dandole todos los argumentos obligatorios que este requiera. Con respecto a los argumentos opcionales, si no enviamos ningún valor para los mismos, tomaran el valor de la expresión que los viene acompañando en la definición del procedimiento. Para llamar a un procedimiento debemos usar la primitiva **CALL**. Veamos un ejemplo:

```
;Dicho metodo nos dibuja un triangulo

TO TRIANGULO :a [:b 5] [:c 5] [:d 5]
  setpc :b
  fd :a
  rt 360 / 3
  setpc :c
  fd :a
  rt 360 / 3
  setpc :d
  fd :a
END

; 1
CALL TRIANGULO 100
; 2
CALL TRIANGULO 100 1
; 3
CALL TRIANGULO 100 1 2
; 4
CALL TRIANGULO 100 1 2 3
```

Veamos que sucede en cada caso:

1. En esta llamada, simplemente llamamos a TRIANGULO pasandole sólamente su argumento obligatorio, por lo tanto dibujará cada lado del triángulo con el color que hace referencia cada argumento opcional en su expresión por defecto.
2. En esta llamada, llamamos a TRIANGULO modificando su primer argumento opcional, por lo tanto dibujará el primer lado del triángulo con el color número 1 de la paleta y los demás lados color que hace referencia cada argumento opcional restante en su expresión por defecto.
3. En esta llamada, llamamos a TRIANGULO modificando su primer y segundo argumento opcional, por lo tanto dibujará el primer lado del triángulo con el color número 1 de la paleta, el segundo lado con el color número 2 de la paleta y el lado restante con el color que hace referencia el último argumento opcional.
4. En esta llamada, llamamos a TRIANGULO modificando todos sus argumentos opcionales, por lo tanto dibujará el primer lado del triángulo con el color número 1 de la paleta, el segundo lado con el color número 2 de la paleta y el tercero con el color número 3 de la paleta.

1.6. Comandos gráficos

La idea es que nosotros tenemos una tortuga virtual, a la cual le vamos dando indicaciones para que vaya caminando y a medida que lo hace deje un rastro. La posición inicial de la tortuga es el centro de la imagen, mirando verticalmente hacia arriba (ángulo de 90°). A continuación veremos los comandos para manejar a la tortuga:

- **FORWARD:**

  FORWARD n
  FD n

  La tortuga da n pasos hacia adelante en la dirección hacia donde mira (1 paso equivale a un pixel).

- **BACK:**

  BACK n
  BK n

  La tortuga da n pasos hacia atrás en la direccion contraria hacia donde mira (el ángulo hacia donde mira la tortuga no cambia).

- **LEFT:**

  LEFT n
  LT n

  La tortuga gira n grados en sentido contra-horario.

- **RIGHT:**

  RIGHT n
  RT n

  La tortuga gira n grados en sentido horario.

- **SETXY:**

  SETXY x y

  Pone a la tortuga en la posición x y de la imágen.

- **SETX:**

  SETX x

  Mueve a la tortuga horizontalmente hasta la coordenada x (la coordenada y se mantiene).

- **SETY:**

  SETY y

  Mueve a la tortuga verticalmente hasta la coordenada y (la coordenada x se mantiene).

- **SETHEADING:**

  SETHEADING n

  Hace que la tortuga mire hacia el ángulo n (en grados).

- **HOME:**

  HOME

  Mueve a la tortuga a la posición y ángulo iniciales.

- **PENUP:**

  PENUP
  PU

  Levanta el lápiz que lleva la tortuga, para que, cuando esta camine, no escriba.

- **PENDOWN:**

  PENDOWN
  PD

  Baja el lápiz que lleva la tortuga, para que, cuando esta camine, escriba.

- **SETPENCOLOR:**

  SETPENCOLOR ⟨color⟩
  SETPC ⟨color⟩

  Cambia el color del lápiz a ⟨color⟩ (en la siguiente sección se especifica qué puede ser ⟨color⟩).

1.7. Colores

Cuando nos referimos a colores, podemos utilizar los que están guardados en la paleta interna (16 colores) o bien utilizar una combinación RGB. Por ejemplo:

```
SETPC 2 ;Cambia el color del lapiz al color 2 de la paleta
SETPC aqua ;Cambia el color del lapiz al color aqua (11) de la paleta
SETPC [128 54 210] ;Cambia el color del lapiz al color 128 54 210 (RGB)
```

A continuación veremos la paleta de colores:

black (0)	blue (1)	green (2)	cyan (3)
red (4)	magenta (5)	yellow (6)	white (7)
brown (8)	tan (9)	forest (10)	aqua (11)
salmon (12)	purple (13)	orange (14)	grey (15)

1.8. Estructuras de control

- **REPEAT**

  REPEAT n [instrucciones]

  Repite n veces las instrucciones dadas entre corchetes. Esta primitiva cuenta con una variable local creada automaticamente llamada **repcount** que nos dice en que numero de repetición estamos.

- **IF**

  IF b [instrucciones]
  (IF b [instrucciones1] [instrucciones2])

  Dependiendo de que valor nos retorna la evaluación de la expresión booleana b, tenemos que:

  - Si devuelve **true** y tenemos solamente una lista de instrucciones la ejecutamos.
  - Si devuelve **false** y tenemos solamente una lista de instrucciones, no las ejecutamos y seguimos la ejecución normalmente.
  - Si devuelve **true** y tenemos dos listas de instrucciones, ejecutamos la primera.
  - Si devuelve **false** y tenemos dos listas de instrucciones, ejecutamos la segunda.

- **IFELSE**

  IFELSE b [instrucciones1] [instrucciones2]

  Dependiendo de que valor nos retorna la evaluación de la expresión booleana b, tenemos que:

  - Si devuelve **true**, ejecutamos la primer lista de instrucciones.
  - Si devuelve **false**, ejecutamos la segunda lista de instrucciones.

- **FOR**

  FOR [forcontrol] [instrucciones]

  [forcontrol] tiene 4 parametros, 3 obliglatorios y 1 opcional:

  1. Nombre de la variable local.
  2. Valor de inicio de la variable local.
  3. Valor limite de la variable local.
  4. Tamaño de salto entre cada iteración.

  El cuarto parámetro es opcional, si no se establece, el salto se definirá automáticamente como 1, si el valor inicial es menor al final, y -1 si el valor inicial es mayor al final. La lista de instrucciones se repetirá hasta que el valor inicial alcance al final.

  Ejemplo de un for:

```
for [i 0 10 2] [
forward i
setpencolor i
]
```

- **WHILE**

  WHILE b [instrucciones]

  Ejecuta la lista de instrucciones hasta que la expresión booleana b evalúe **false**. Primero evalúa la expresión booleana y luego decide si ejecutar o no las instrucciones.

- **DO.WHILE**

  DO.WHILE [instrucciones] b

  Ejecuta la lista de instrucciones hasta que la expresión booleana b evalúe **false**. Primero ejecuta la lista de instrucciones y luego evalúa la expresión booleana, por lo tanto habrá al menos una ejecución.

- **UNTIL**

  UNTIL b [instrucciones]

  Ejecuta la lista de instrucciones hasta que la expresión booleana b evalúe **true**. Primero evalúa la expresión booleana y luego decide si ejecutar o no las instrucciones.

- **DO.UNTIL**

  DO.UNTIL [instrucciones] b

  Ejecuta la lista de instrucciones hasta que la expresión booleana b evalúe **true**. Primero ejecuta la lista de instrucciones y luego evalúa la expresión booleana, por lo tanto habrá al menos una ejecución.

2. Estructuras de datos utilizadas

A continuación se mostrarán todas las estructuras de datos utilizadas para cada una de las partes de la implementación del compilador. Las estructuras de datos están alojadas en el archivo **AST.hs**.

2.1. Puntos en el plano

Para ir calculando los puntos de la imagen que deben ser pintados utilizamos el tipo Coord, que no es más que una tupla que contiene un par de Doubles. Los puntos habitualmente se van calculando paso por paso. El motivo de por que usamos flotantes de presición doble es para tener más exactitud a la hora de ir calculando los puntos a dibujar.

2.2. Expresiones aritméticas

La estructura de datos que nos permite formar los AST de las expresiones aritméticas es el siguiente:

```
data NumExp = Const Double
            | Var Variable
            | BinOp (Double -> Double -> Double) NumExp NumExp
            | UnOp (Double -> Double) NumExp
            | Div NumExp NumExp
```

Por simplicidad no diferenciamos entre enteros y flotantes, sino que tratamos cualquier entidad numérica como simplemente un número. Tenemos constructores para constantes y variables.

En lugar de usar un constructor por cada operación binaria/unaria optamos por que el costructor en si mismo pueda recibir la funcion que necesitamos, esta función de establece en el momento del parseo. La operación de la división la tenemos separada de las operaciones binarias restantes para poder tener manejo de errores sobre la division por 0.

2.3. Expresiones booleanas

La estructura de datos que nos permite formar los AST de las expresiones booleanas es el siguiente:

```
data BoolExp = C Bool
             | VarB Variable
             | BinOpN (Double -> Double -> Bool) NumExp NumExp
             | BinOpB (Bool -> Bool -> Bool) BoolExp BoolExp
             | Not BoolExp
```

Tenemos constructores para constantes y variables, y usamos el mismo método que con las expresiones aritmética para las operaciones binarias.

2.4. Primitivas

Para las primitivas utilizamos el tipo de datos **Comm**, el cual tiene un constructor por cada primitiva implementada.

2.5. Colores

Para los colores tenemos la estructura **Colors**, la cual tiene un cosntructor por cada color de la paleta, y un costructor **Custom Color**, el cual nos permite utilizar cualquier color que este en formato RGB.

2.6. Definición de procedimientos

Para representar la definición de procedimienots con el comando **TO** utilizamos la siguiente estructura de datos:

```
type OpArg = (Variable, Val)
type Function = (String, [Variable], [OpArg], [Comm])
```

El primer componente de la tupla representa el nombre del procedimiento, el segundo la lista de argumentos **obligatorios**, el tercero la lista de argumentos opcionales (los argumentos opcionales estan representados por el tipo **OpArg**, el cual es una tupla en donde el primer componente es el nombre del argumento y el segundo componente es un valor, que puede ser tanto numérico como booleano), y por último, la lista de comando que debe ejecutar dicho procedimiento.

3. Evaluación

Para la evaluación del código, utilizamos la mónada **State** ligeramente modificada, agregando mas cosas que solamente un entorno para las variables. Veamos cómo se define:

```
data Val = VN Double | VB Bool
type Env = [(Variable, Val)]
type EnvF = [Function]

type TurtleThings = (Coord, Double, Color, (Int, Int), Bool, Color)

type ImagePix = (IntCoord, Color)

data ImgDesc = Ip ImagePix
             | Il IntCoord IntCoord Color

newtype State a = State { runState :: Env ->
                                      EnvF ->
                                      TurtleThings ->
                                      [ImgDesc] ->
                                      Env -> Either Error (a, Env, EnvF, TurtleThings, [ImgDesc], Env) }
```

Env cumple la función de entorno para las variables globales, **EnvF** es un entorno para los procedimientos declarados en el código (la información necesaria de cada uno esta encapsulada en el tipo de dato **Function**).

TurtleThings es un tipo de dato que contiene toda la información necesaria para la poder ir dibujando mientras se lleva a cabo la compilación, la información que contiene esta tupla es: posición en donde se encuentra la tortuga en ese momento, ángulo hacia donde mira, color de escritura, tamaño de la imagen, si el lápiz está apoyado o no en la hoja y el color de fondo de la imagen.

[**ImgDesc**] es donde vamos armando la imagen a medida que vamos compilando, es una lista que contiene una descripción de la imágen la cual después se creará con la librería gráfica GD. Cada elemento de la descripción puede ser una de las siguientes cosas:

- Ip ImagePix : Representa un punto concreto de la imagen, con su color correspondiente.
- Il IntCoord IntCoord Color : Representa un segmento entre dos coordenadas de la imagen, con su color correspondiente.

Por último, el último **Env** se utilizó para debuggear, por lo tanto en la versión final no cumple ningún rol importante.

3.1. Evaluación de variables locales

Para poder utilizar variables locales (tanto en procedimientos, como en estructuras de control como for), en cada función de evaluación (de comandos, expresiones numéricas, etc) agregamos como argumento un entorno. Al momento de buscar el valor de una variable, primero lo hacemos sobre este entorno local pasado por argumento, y, si no lo encontramos, recien ahí nos fijamos en el entorno global.

3.2. Construcción de la imagen final

Una vez que el proceso de evaluación termina, obtendremos una descripción completa de la imagen. Dicha descripción consta de:

- Tamaño de la imagen.
- Color de fondo.
- Pixeles o líneas a ser dibujadas.

A partir de esta descripción utilizaremos la libreriía gráfica GD para crear la imágen. Una vez hecho esto el proceso de compilación se considera terminado.

4. Organización de archivos y software de terceros

El código de fuente esta dividido de la siguiente forma:

- AST.hs : Contiene todas las estructuras de datos necesarias.
- Parser.hs : Contiene el parser.
- Evaluation.hs : Contiene los evaluadores monádicos.
- Mains.hs : Modulo principal que enlaza todo lo demás.

Para el parseo se utilizó la librería Parsec, mientras que para la creación de las imágenes se utilizó la librería GD. Un fragmento de la descripción del lenguaje logo de la primera página fue extraido de wikipedia.

5. Instalación y modo de uso

Para poder compilar el software, primero debemos instalar las librerías Parsec y GD. Veamos un ejemplo de qué comandos habría que usar en ubuntu

```
~$ sudo apt-get install libgd2-xpm-dev
~$ cabal install gd
~$ cabal install parsec
```

Luego para compilar el código

```
ghc --make -O2 Main.hs
```

Para utilizar el compilador solo hay que ejecutarlo de la siguiente manera:

```
~$ ./Main fuente.lgo nombresalida
```

Se agregará automáticamente la extensión png al archivo de salida.