Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Trabajo Práctico final
Análisis de Lenguajes de Programación
WLogo

Implementacion de un evaluador de Logo

Autor:
Wirzt Santiago

Febrero 2022

Contenidos

1 Motivación

La idea del proyecto surgió porque en mis clases de programación en la escuela tuve mi primer acercamiento a la programación con el Logo Writer. Un evaluador de Logo en el que debíamos copiar dibujos otorgado por la profesora.

Logo es un lenguaje de programación de alto nivel, en parte funcional, en parte estructurado; de muy fácil aprendizaje, razón por la cual suele ser el lenguaje de programación preferido para trabajar con niños y jóvenes. Wikipedia

Los evaluadores de Logo suelen contar con una interfaz gráfica y una de consola que funcionan en simultáneo. Con comandos ingresados por la consola damos órdenes a una tortuga que al moverse por la pantalla deja un rastro de sus movimientos, así pudiendo crear dibujos con código.

2 Decisiones de diseño

Al inicio del trabajo intenté implementar el lenguaje Logo completo pero se volvió algo muy complejo y tuve que disminuir modificarlo para noexceder el alcance del proyecto. Uno de los cambios más importantes es la eliminación de las listas, en mi lenguaje solo se puede almacenar valores de a uno.
Otro cambio fue la unión de los valores numéricos y booleanos, tomé un enfoque similar al de el lenguaje C. El 0 es falso y el resto de números es verdadero.
El evaluador trabaja todos los números como flotantes. Al trabajar con Flotantes los errores de redondeo son frecuentes así que reemplacé la igualdad por una distancia inferior a un Épsilon, en este caso es 0.00005. Esto afecta un poco a lo dicho anteriormente, evalúan a falso los números que se encuentran entre -0.00005 y 0.00005.
La mayor parte del loop del programa se lleva a cabo por la función simulateIO de la librería Gloss. Esto llevó a que para poder leer de consola al mismo tiempo que estoy dentro de la "simulación", debí llevar la lectura de consola a un hilo a parte y mis hilos se comunican con variables de IO de tipo MVar.
Para que la parte gráfica sea más entretenida, no puedo evaluar todos los comandos de una vez ya que una operación for muy larga podría hacer que haya muchos tiempo entre una actualización y otra y hace que el dibujo cambie mucho de un momento al otro. Por lo que evalúo los comandos de a "pasos", donde algunos son 1 paso y otros generan más pasos a evaluar.

3 Gramática

3.1 Sintaxis Concreta

Definimos la sintaxis concreta del DSL a partir de la siguiente gramática:

```
Supongo String y Float ya definidos

Comm ::= Comm Comm
       | forward Exp | fd Exp -- Avanza la tortuga Exp distancia
       | back Exp | bk Exp -- Retrocede la tortuga Exp distancia
       | right Exp | rt Exp -- Gira la tortuga Exp grados hacia la derecha
       | left Exp | lt Exp -- Gira la tortuga Exp grados hacia la izquierda
       | clearscreen | cs -- Limpia la pantalla y devuelve el estado a su valor original
       | clean -- Solo limpia el dibujo, el resto de elementos quedan en su lugar
       | penup | pu -- Levanta el lápiz, la tortuga no deja rastro al moverse
       | pendown | pd -- Baja el lápiz, la tortuga si deja rastro
       | hideturtle | ht -- No muestra la tortuga en pantalla
       | showturtle | st -- Muestra la tortuga en pantalla
       | home -- La tortuga camina hacia el centro y vuelve a mirar hacia arriba
       | setx Exp | sety Exp | setxy Exp Exp
       -- Coloca manualmente la tortuga en la posición dada (Depende de la función)
       y marca el recorrido en el dibujo
       | setheading Exp | seth Exp -- Apunta la tortuga a Exp grados
       | repeat Exp '[' Comm ']' -- Repite los comandos Comm, Exp veces
       (aplica piso en caso de que no sea un número entero)
       | print Exp -- Muestra en consola el valor de Exp
       | print '"'String -- Muestra en consola el String
       | setcolor Exp -- Modifica el color del recorrido de la tortuga a un color predefinido
       entre 0 y 15
       | setcustomcolor Exp Exp Exp -- Modifica el color del recorrido de la tortuga a un
       color personalizado con sus valores RGB (entre 0 y 255)
       | if Exp '[' Comm ']' '[' Comm ']' -- Operador IF
       | to String Strings Comm end -- Definición de funciones, reemplaza en Comm las
       variables dadas en Strings
       | make '"'String Exp -- Almacena una variable String con un Valor Exp en memoria.
       Si se repite el nombre sobrescribe al anterior
       | for '[' '"'String Exp Exp ']' '[' Comm ']' -- Operación FOR con Delta = 1
       | for '[' '"'String Exp Exp Exp ']' '[' Comm ']' -- Operación FOR con un Delta dado
       | wait Exp -- Pausa la ejecución Exp milisegundos
       | while Exp '[' Comm ']' -- Operación WHILE
       | do.while '[' Comm ']' Exp -- Operación DO WHILE
       | String ExpSeq -- Llamada a una función ya definida
       | scale Exp -- Modifica la escala del dibujo mostrado en la ventana, por defecto es 5
       | arc Exp Exp -- Crea un arco de radio y ángulos dados por el usuario a partir del
       ángulo actual de la tortuga
       | label '"'String -- Escribe un texto en la ventana (No esta implementado perfecto)
       | setlabelheight Exp -- Aumenta o disminuye el tamaño del texto para label
       | undo -- Deshace solo una iteración de dibujado (No deshace definiciones y la tortuga
       se mantiene donde quedó)
       | skip -- Operación vacía
       | '(' Comm ')'

Strings ::= String
          | String Strings

ExpSeq ::= Exp
```

```
        | Exp ExpSeq

Exp ::= Float -- Valor numérico
      | '-' Exp
      | xcor | ycor -- Coordenadas x o y de la tortuga
      | heading -- Valor del ángulo actual de la tortuga
      | towards Exp Exp -- Valor de ángulo que necesita la tortuga para apuntar a la posición dad
      | ':'String -- Busca el valor de una variable en memoria
      | Exp '+' Exp
      | Exp '-' Exp
      | Exp '*' Exp
      | Exp '/' Exp
      | readword -- Lee una Exp de consola
      | if Exp Exp Exp -- Operación IF a nivel Exp
      | Exp '>' Exp
      | Exp '<' Exp
      | Exp '=' Exp
      | Exp '<=' Exp
      | Exp '>=' Exp
      | Exp '!=' Exp
      | Exp '&&' Exp
      | Exp '||' Exp
      | not Exp
      | true
      | false
      | random Exp -- Obtiene un valor aleatorio entre 0 y Exp
      | '(' Exp ')'
```

3.2 Notación

Un String no puede contener espacios y los nombres de funciones y variables no pueden empezar con un número Si un String es precedido por : o ", no debe haber un espacio entre ellos
No existe distinción entre caracteres en mayúscula o minúscula.

4 Resolución

4.1 Parseo

El parseo se llevó a cabo usando el paquete Happy que me permitió definir ordenes de precedencia y asociatividad fácilmente. El código del parser se encuentra en el archivo `src/LogoPar.y`.
Siguiendo la documentación que proporciona la herramienta logré hacer control de errores con un parser monádico y seguimientos de líneas/columnas con un lexer mejorado.

4.2 Uso de las Mónadas

En el archivo `src/MonadLogo.hs` se encuentra la definición de la mónada usada y de todos sus funciones auxiliares. La monada usada esta formada de 2 transformadores de mónadas sobre la mónada IO:

- ExceptT – Nos otorga la función throwError para elevar errores durante la ejecución sin terminar la ejecucion del programa.
- StateT – Nos provee las funciones de get, set y modify para llevar un estado global durante la ejecución de nuestro programa.
- IO – Utilizo la mónada IO para poder acceder a las funciones de la variable de entorno MVar ya que la entrada y salida de consola es trabajada por el otro hilo.

4.3 Evaluación

El evaluador del proyecto se encuentra en `src/Eval.hs`, allí defino las funciones de eval y runExp que se encargan de evaluar todo el código ingresado y funciones auxiliares que disminuyen la cantidad de código repetido. Estas funciones reciben como argumento el entorno local actual (Una lista de Expresiones) y el comando o expresión a evaluar.
Para aclarar, todas las expresiones se evaluán de forma bigStep, donde la ejecución no termina hasta llegar a un valor final. En cambio, los comandos se asemejan a una ejecución littleStep, donde un comando puede terminar de ejecutarse o generar otro entorno para un comando que debe ejecutarse. El valor de retorno de la función `eval` es de la forma [([Exp], Comm)] que representa una lista de comandos con su correspondiente entorno local.

4.4 Otros Archivos

- En `src/Common.hs` se definen las estructuras de datos usadas para representar nuestro lenguaje, así como su instancia de Show.
- En `src/Lib.hs` se implementan las funciones útiles para usar en el resto del programa.
- En `src/GlobalEnv.hs` se define la estructura del estado que se llevara en la mónada estada y sus valores iniciales.
- En `app/Main.hs` se encuentran las funciones que permiten al programa correr. Estas se encargan de:
  - Parsear los parámetros dados al momento de ejecutar el programa.
  - Iniciar las variables IO y el hilo que se ocupará de la consola interactiva.
  - Las funciones ocupadas de llevar el código del hilo de consola.
  - Y la llamada a la función principal `simulateIO` que es la encargada de mantener el loop de ejecución del programa, ejecutando cada cierta cantidad de milisegundos a la función `step` que avanza el estado un paso y la función `env2pic` que convierte el estado actual en una imagen para mostrar en pantalla.

5 Instalación y Uso

5.1 Manual de instalación

Estas instrucciones asumen que el sistema operativo utilizado es Linux (No se probó en Windows) y que el instalador de paquetes es apt

- Primero tener los paquetes del sistema actualizados

```
sudo apt update
sudo apt upgrade
```

- Se necesita instalar git y make

```
sudo apt install git make
```

- Luego clonar el repositorio git del tp

```
git clone https://github.com/swirzt/WirztLogo.git
```

- Ingresamos a la carpeta y ejecutamos:

```
make env -- Se encarga de instalar todos los paquetes necesarios para que el programa
funcione, requiere acceso root. Se puede acceder a los comandos en el archivo Makefile.
```

- Cerrar y abrir la terminal para actualizar el PATH de los programas instalados
- Volver a la carpeta y ejecutar:

```
make compile -- Ejecuta stack build con las banderas para que funcione correctamente.
```

- Puede que durante el proceso se pida confirmación para algunas acciones.

5.2 Manual de uso

El programa se ejecuta utilizando el comando:

```
stack run
```

Esto inicia la consola y abre una ventana gráfica donde se encuentra la tortuga. Para interactuar con la tortuga se deben escribir los comandos deseados en la consola, estos se evaluaran y se mostrarán en la ventana gráfica.
Se pueden pasar argumentos al programa mediante banderas al momento de ejecutarlo, colocándolos luego de $--$.
Los argumentos posibles son:

- -f :: Abre la parte gráfica en pantalla completa, utilizar esta opción hace redundantes las dos siguientes (Solo recomendado si se corre con un archivo precargado ya que se hace difícil acceder a la consola para escribir los comandos)
- -h Num :: Alto de la ventana gráfica en píxeles (Por defecto es 300)
- -w Num :: Ancho de la ventana gráfica en píxeles (Por defecto es 300)
- -r Num :: Velocidad de refresco (Por defecto es 60 veces por segundo)
- -m i/o :: Modo de funcionamiento del programa (Se explica más adelante)

- NombreDeArchivo.logo :: Si no se da una bandera se toma el argumento como un archivo con comandos para evaluar

Un ejemplo de ejecución podría ser:

```
stack run -- -w 1280 -h 720 -r 2000 testerTowards.logo
```

El programa puede funcionar de 2 formas:

- Modo interactivo **-m i** :: Este es el modo usual de evaluación, todo los inputs son evaluados uno a uno y mostrados en la ventana gráfica.
- Modo compilación **-m o** :: Este modo requiere que se le den al menos 2 nombres de archivo en la línea de comandos, el programa evaluará los comandos y la guardará el resultado con el nombre del último archivo ingresado, la extensión del archivo determina de que forma se guardara el dibujo (Gif o PNG). No hace uso de la parte gráfica.

Por último, cuando se está en modo interactivo, existen instrucciones para interactuar con el entorno:

- :? - Muestra el menú de ayuda
- :q - Termina la ejecución
- :v - Lista los valores guardados de variables en el entorno
- :c - Lista las definiciones guardadas de comandos en el entorno
- :l archivo.logo - Carga un archivo con comandos
- :sg archivo.gif - Guarda el dibujo actual en un GIF animado con el nombre dado
- :sp archivo.png - Guarda el dibujo actual en una imagen PNG con el nombre dado

6 Mejoras a futuro

- Agregar soporte de listas. Implica muchos cambios en la gramática y de las funciones, además, las listas de Logo son Heterogéneas, algo que se hace un poco difícil en Haskell y se necesitan librerías aparte.
- Poder cambiar grosor del trazo de la tortuga
- Poder pintar el fondo o secciones de el. Tal vez requiera cambiar la interfaz gráfica ya que al ser dibujos vectoriales se complica un poco determinar que áreas pintar
- Cambio de formas de la tortuga
- Mejor parseo del ingreso a consola, por ahora es un chequeo de strings y puede generar algún comportamiento imprevisto.

7 Bibliografía

- Página donde obtuve la mayoría de los comandos de Logo y utilicé su evaluador para entender como deberían funcionar algunos comandos:

  Turtle Academy

- Librería gráfica Gloss

- El siguiente link es una pregunta que realicé en stackoverflow, es donde me comentan que simulateIO no puede ser bloqueada y el uso de variables IO.

  StackOverflow

- MVar en Control.Concurrent

- La mónada usada es similar a la mónada del Compilador de la materia compiladores.

  Mónada

- Librería de parseo de argumentos por consola