Intérprete Interactivo para Enseñar Lógica Algorítmica: Un Enfoque Basado en Lightbot

Pedroza Palomar, Oriana[1]; Hernández, Juan[1]

[1] *Escuela de Ingeniería de Sistemas, Facultad de Ingeniería, Universidad de Los Andes, Mérida, Venezuela.*

Resumen: Este documento describe el desarrollo de un intérprete basado en un lenguaje inspirado en el juego educativo "Lightbot". El proyecto se divide en tres etapas fundamentales: el analizador léxico (escáner), el analizador sintáctico (parser) y el intérprete. Cada componente del sistema fue diseñado para interpretar un conjunto de instrucciones que permiten controlar un robot simulado; las instrucciones incluyen movimientos, giros, bucles y activación de luces, así como se evidencia en el juego original.

Palabras clave: analizador léxico, analizador sintáctico, lenguaje de programación, interprete, instrucciones.

INTRODUCCIÓN

El aprendizaje de la programación a través de entornos visuales es una técnica utilizada en múltiples plataformas, juegos como *Lightbot* enseñan a los usuarios los fundamentos de la lógica algorítmica mediante la resolución de puzzles. En este contexto, se propone el desarrollo de un intérprete para el lenguaje de programación no tipado diseñado para enseñar lógica algorítmica fundamental de manera interactiva y atractiva, inspirado en el juego *Lightbot*. Este lenguaje permite a los usuarios controlar un robot mediante una serie de comandos para realizar tareas específicas y resolver problemas lógicos.

La implementación del intérprete comienza con el desarrollo del scanner, también conocido como analizador léxico, que constituye la primera etapa del proceso de compilación. Durante esta fase, el scanner lee el código fuente y convierte la entrada en tokens, basados en un conjunto de reglas definidas para el lenguaje. Cada token representa un componente sintáctico del lenguaje, como palabras clave, identificadores, números, operadores y delimitadores. Esta etapa es crucial para transformar el código fuente en una forma que puede ser procesada por las etapas siguientes del intérprete.

METODOLOGÍA

El proyecto se implementó en C++ utilizando herramientas de análisis léxico y sintáctico como Flex y Bison. La arquitectura del sistema se dividió en tres componentes principales: el scanner, que tokeniza las entradas del usuario; el parser, que valida la gramática del programa; y el intérprete, que ejecuta los comandos.
Los comandos principales incluyen:

- Move: Desplaza al robot una posición hacia adelante.
- TurnLeft: Gira el robot 90 grados a la izquierda.
- TurnRight: Gira 90 grados a la derecha.
- Loop: Ejecuta un bloque de comandos repetidamente, según el número de iteraciones especificadas.
- LightUp: Activa una luz en la posición actual del robot.

SCANNER (ANALIZADOR LÉXICO)

En esta etapa se procede a definir cuáles serán los tokens que se utilizarán en las etapas subsiguientes del proceso de compilación. Estos tokens representan los elementos básicos del lenguaje que se necesitan reconocer como movimientos, giros, encendido de luz, identificadores, enteros, bucles, procedimientos, entre otros.

En el archivo scanner.flex se definen las reglas para reconocer correctamente cada token, como por ejemplo:

- MOVE: [Mm][Oo][Vv][Ee]
- LOOP: [Ll][Oo][Oo][Pp]
- TURN_LEFT: [Tt][Uu][Rr][Nn][_][Ll][Ee][Ff][Tt]

Retornando así los siguientes tokens que representan comandos básicos que permiten al robot realizar acciones específicas como por ejemplo moverse siempre de frente, voltear a la izquierda o derecha y encender la luz para finalizar. Así mismo se incluye un token para representar la estructura de control como el bucle, y el procedimiento con su respectivo llamado.

- TOKEN_MOVE
- TOKEN_TURN_LEFT
- TOKEN_TURN_RIGHT
- TOKEN_LIGHT_UP
- TOKEN_LOOP
- TOKEN_PROCEDURE
- TOKEN_CALL

Los siguientes tokens ayudan a estructurar el código y a separar los elementos de las instrucciones, son conocidos como los delimitadores ‘()’, ‘{}’ y los operadores ‘,’ ‘=’.

- TOKEN_LEFT_PAREN
- TOKEN_RIGHT_PAREN

- TOKEN_LEFT_BRACE
- TOKEN_RIGHT_BRACE
- TOKEN_COMMA
- TOKEN_EQUAL

No obstante, se definen los tokens TOKEN_EOF, TOKEN_INT, TOKEN_IDENTIFIER, los cuales son fundamentales para identificar el final del archivo, los números enteros y los identificadores del lenguaje. El escáner maneja la eliminación de espacios en blanco y comentarios, mejorando la claridad del código fuente para el parser.

PARSER (ANÁLISIS SINTÁCTICO)
VALIDADOR

El analizador sintáctico se encarga de verificar que los tokens generados en la etapa anterior (scanner) sigan la estructura gramatical del lenguaje. En el archivo parser.bison, se define la gramática del lenguaje, la cual sigue el patrón de un conjunto de comandos y procedimientos. Un ejemplo sencillo es que un procedimiento como main() puede contener comandos de movimiento, giros y bucles.

Esta etapa es crucial para detectar errores sintácticos, como la falta de paréntesis o llaves mal emparejadas, proporcionando mensajes claros al usuario sobre el problema detectado.

El archivo parse.bison tiene la siguiente estructura:

Definiciones y Declaración de Tokens:

Se define la función `yylex()` que es del scanner y `yyerror()` para el manejo de los errores. También se declaran los tokens que el scanner reconoce y pasa a esta etapa (parser), como `%token TOKEN_INT`, `%token TOKEN_MOVE`, `%token TOKEN_TURN_LEFT`, entre otros.

Reglas gramaticales:

Por su parte, las reglas gramaticales definen cómo los tokens se agrupan para formar construcciones válidas en el lenguaje. En este caso, la regla principal es un programa (**program**) que comienza con un procedimiento cuya palabra clave es PROCEDURE.

```
program : procedure;
```

La siguiente regla es la definición del procedimiento (**procedure**), el cual está compuesto por su palabra clave PROCEDURE, seguido de un identificador, paréntesis y un bloque de comandos delimitado por {}. Este bloque puede contener varias acciones o comandos que el robot debe ejecutar.

```
procedure :  TOKEN_PROCEDURE TOKEN_IDENTIFIER TOKEN_LEFT_PAREN
             TOKEN_RIGHT_PAREN TOKEN_LEFT_BRACE commands
             TOKEN_RIGHT_BRACE;
```

La última regla es sobre los comandos (**commands y command**), la cual permite que un procedimiento contenga uno o más comandos (acciones que el robot puede realizar). Los comandos incluyen movimientos, giros, bucles y llamadas a procedimientos.

```
commands :  commands command
         | command;
```

```
command :  TOKEN_MOVE TOKEN_LEFT_PAREN TOKEN_RIGHT_PAREN
         | TOKEN_TURN_LEFT TOKEN_LEFT_PAREN TOKEN_RIGHT_PAREN
         | TOKEN_TURN_RIGHT TOKEN_LEFT_PAREN TOKEN_RIGHT_PAREN
         | TOKEN_LIGHT_UP TOKEN_LEFT_PAREN TOKEN_RIGHT_PAREN
         | TOKEN_LOOP TOKEN_LEFT_PAREN TOKEN_INT
           TOKEN_RIGHT_PAREN TOKEN_LEFT_BRACE commands
           TOKEN_RIGHT_BRACE
         | TOKEN_CALL TOKEN_IDENTIFIER TOKEN_LEFT_PAREN
           TOKEN_RIGHT_PAREN;
```

Un ejemplo sencillo que cumpla con la estructura es el siguiente:

```
PROCEDURE identificador(){
    commands
}
```

En cuanto al manejo de errores, la función `yyerror()` proporciona un mensaje claro cuando se detecta un error, indicando la línea donde ocurrió el problema.

Con respecto a la validación, es necesario analizar el archivo de entrada (ejemplo1.txt), para verificar si cumplen con las reglas sintácticas definidas en la gramática del lenguaje. El objetivo de esta etapa es procesar el archivo.txt, hacer uso del parser.bison agregando el archivo a `yyin`, luego `yyparse()` lo procesa y corrobora si cumple con la gramática.

Dependiendo del retorno de `yyparse()` se imprime un resultado que cual indique sí: el análisis fue exitoso, entonces el validador muestra “Parser successfull”, o si hubo errores sintácticos se muestra “Parser failed!”.

INTERPRETE

El intérprete se encarga de ejecutar el árbol de sintaxis abstracta (AST) generado por el parser. En esta etapa se utiliza la misma estructura del scanner, solo agregando una variable global para manejar el valor de los números enteros (int int_value). En cuanto al archivo parser.bison se modifica ya que es necesario agregar ciertas instrucciones para cada regla de modo que se pueda construir un árbol de comandos, el cual incluye objetos como `Move, TurnLeft, Loop`, entre otros, que se organizan en una lista de comandos (CommandList).

El objeto `parser_result` es el resultado final del análisis sintáctico. Este objeto es de tipo `Command*`, lo que significa que puede apuntar a cualquier comando o lista de comandos (`CommandList`) que haya sido construido durante el proceso de parsing; cuando el parser termina, `parser_result` contendrá la raíz del árbol de comandos, que puede ser un `Procedure` o una lista de comandos.

Las clases de comandos están definidas en commands.hpp y commands.cpp, aquí se implementa la lógica de ejecución. Estas clases heredan de una clase base `Command`, que define las operaciones principales de cada instrucción (el método execute y el método destroy). En cuanto a la clase `CommandList`, es una lista de múltiples comandos que son ejecutados uno por uno.

De la misma forma se encuentran las clases básicas que son `Move, TurnLeft, TurnRight, LightUp`, donde cada una

de ellas implementa el método `execute()` que permitirá que se realicen acciones específicas del robot. La clase `Loop` es especial, esta ejecuta una lista de comandos (`CommandList`) un número determinado de veces (`iterations`). El método `execute()` de la clase `Loop` recorre este bucle y ejecuta la lista de comandos tantas veces como `int_value` lo indique.

Finalmente, la clase `Procedure` almacena una lista de comandos que se ejecutan cuando se llama al procedimiento. Y `ProcedureCall` se utiliza para invocar un procedimiento por su nombre, se busca el procedimiento correspondiente y se ejecuta su lista de comandos.

CONCLUSIONES

La etapa de análisis léxico es esencial para dividir el código fuente en tokens significativos que representen los comandos del robot. La herramienta Flex permitió establecer reglas claras para identificar estos tokens, asegurando una correcta interpretación de los comandos básicos, los identificadores, los números enteros y los delimitadores. Esta etapa proporcionó una fundación para el análisis sintáctico, utilizando un escáner para filtrar cualquier comentario y espacio en blanco para asegurarse de que la entrada fuera una cadena significativa.

El parser se ocupó de verificar si en el scanner los tokens que guardamos siguieron las reglas gramaticales precedentes definidas en Bison. Ayudó en la identificación de los errores sintácticos y, de esta manera, aseguró que todos los programas se compusieron estructuralmente correctamente antes de la ejecución. La última etapa era crítica para validar la lógica de los comandos, por esto en el intérprete se crea el árbol de comandos generado por el parser donde describe todas las acciones que el robot debe realizar, y se ejecuta este árbol siguiendo la lógica definida en las clases de comandos.

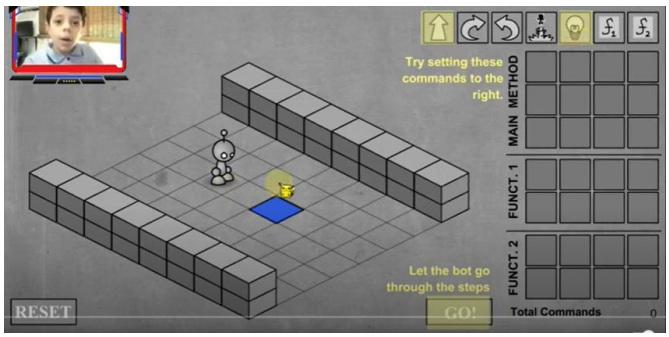

Ilustración 1. Imagen de referencia del juego LigthBoot

REFERENCIAS

Lightbot. (n.d.). *Lightbot: Learn to code by solving puzzles*. Retrieved July 18, 2024, from https://lightbot.com/

Mujica, Alejandro. (2024). Lecture01. Obtenido de: https://docs.google.com/presentation/d/1OkOJO4Ppw0JdPZ6m6bZC6EDXWgsUrmnqe5ai-GWzb4A/edit?usp=sharing. (Julio, 2024).

Mujica, Alejandro. (2024). Lecture02. Obtenido de: https://docs.google.com/presentation/d/1OkOJO4Ppw0JdPZ6m6bZC6EDXWgsUrmnqe5ai-GWzb4A/edit?usp=sharing. (Julio, 2024).

Mujica, Alejandro. (2024). Lecture03. Obtenido de: https://docs.google.com/presentation/d/1OkOJO4Ppw0JdPZ6m6bZC6EDXWgsUrmnqe5ai-GWzb4A/edit?usp=sharing. (Julio, 2024).

Rasmos, R., Salas, J., & Hernández, J. (2024). *Programming languages: Programmer's playground project*. Retrieved July 18, 2024, from https://docs.google.com/document/d/1MKd55Z-GijVhqKZsJWUnXSrMCQTPV6o8EJCxd81vUcs/edit#heading=h.5qhyiwmbjyq2