UACM
Universidad Autónoma de la Ciudad de México
Nada humano me es ajeno

COLEGIO DE CIENCIA Y TECNOLOGÍA

LICENCIATURA EN INGENIERÍA DE SOFTWARE

Diseño y desarrollo de un software integrado (IDE, compilador y máquina virtual), para la enseñanza de la lógica de programación a partir de edades de 15 años, por medio de la creación propia de un lenguaje de programación de rutas instruccionales

T E S I S

QUE PARA OPTAR POR EL TÍTULO DE

LICENCIADO EN INGENIERÍA DE SOFTWARE

P R E S E N T A :

JONATHAN GARCÍA GIL

D I R E C T O R

MTRO. ADALBERTO ROBLES VALADEZ

Ciudad de México, mayo de 2023.

UACM
Universidad Autónoma de la Ciudad de México
Nada humano me es ajeno

CA Coordinación Académica

SISTEMA BIBLIOTECARIO DE INFORMACIÓN Y DOCUMENTACIÓN

UNIVERSIDAD AUTÓNOMA DE LA CIUDAD DE MÉXICO
COORDINACIÓN ACADÉMICA

RESTRICCIONES DE USO PARA LAS TESIS DIGITALES

DERECHOS RESERVADOS©

Contenido

Nota: En este documento existen referencias a apéndices, éstos se pueden encontrar en un disco al final de este trabajo escrito.

Agradecimientos

Este trabajo escrito es suficientemente largo como para no agradecer a cada persona que indirectamente puso su granito de arena para que yo me encuentre hasta dónde estoy.

Comenzaré agradeciendo a cada uno de mis profesores del semestre de integración, que para los que no saben, es el semestre introductorio con el que inicias en la universidad. Gracias a María Eugenia Covarrubias, Javier Vargas que me enseñó la cosa más valiosa que me ayudó en toda mi carrera: La multiplicación entre fracciones con un método distinto al llamado “Sándwich” y a Eduardo Delgado. Este último que siempre me reconoció hasta el último semestre que tomé clase.

Seguiré con mis profesores de ciclo básico del plantel Cuautepec, Alberto Ceciliano, Rocío Sánchez, Manuel Chaidez, Erick Muñoz, Claudia Domínguez, Israel Gallegos, Carlos Avendaño (que siempre trató de hacer sus clases menos pesadas), Edith Leal (la única que me aplicó un examen sorpresa), y en especial a Juan Manuel Reyes que fue el único profesor que siempre me dio la hora y media de clase (3 horas en algunos casos), siempre puntual, ni más ni menos, y que vuelve clases pesadas (mecánica/física) en algo muy interesante.

No olvido a mis profesores de ciclo básico del plantel Centro Histórico Iván Orencio, Alberto Quiroz, José Luis Lugo, Gerardo González, José Gregorio, y Martín López.

Mis profesoras de inglés, Amalia Elena, María de los Ángeles Escobar y Gabriela Guevara. La academia de inglés de la UACM tiene los mejores profesionistas, y ojalá en algún futuro se les permita preparar aún más a los estudiantes (a exámenes de certificación internacional).

A mi único profesor de San Lorenzo Tezonco, Marcos Chimil.

Y mis profesores de ciclo superior del plantel Cuautepec Violeta Jiménez, Salvador Ortega, Víctor Álvarez, Antonio Ramírez, Magnolia García (estudiante egresada de UACM), Elio Santiago, Sandy Mendieta, Jorge Wals y Emmanuel Vite. En especial a Gerardo Hernández (hoy profesor de base), Federico Juárez, Vivanco Gallardo y claro, Adalberto Robles mi asesor de tesis, amigo y gran responsable de lo que soy como profesionista.

A mis colegas Paloma Martínez (la comadre), Miriam Mecalco, Roberto Gil, Christian Alarcón, Ana Montiel y los que me faltan. En especial a Gerardo Baltazar, gracias por hacer mis últimos semestres los mejores de mi carrera. Aprovecho para disculparme con algunos compañeros por no haber sabido trabajar en equipo, lo entendí muy tarde.

A mis amigos de toda la vida: Jatsiri Pizarro, Jesús Rodríguez, Jatsiri Hernández, Alondra Varela, David Alvarado, Francisco Alvarado, Vanesa González y los que me faltan, gracias por su apoyo en estos años.

Y claro, mi familia. Por ser pacientes mucho tiempo para este momento. Mi mamá y mi tía, 2 mujeres trabajadoras que me ayudaron en todos los años de mi carrera. Mi eterna gratitud.

Gracias a la Universidad Autónoma de la Ciudad de México, siempre estaré con la frente en alto, orgulloso de esta institución, por todas las experiencias que viví dentro de ella. **Autonomía, Educación y Libertad.**

0 Introducción

Quiero comenzar platicando un poco de mi experiencia cuando yo aprendí a programar. Una de mis materias preferidas de primer semestre en la licenciatura fue la materia de Introducción a la programación, cuando yo entré a la universidad no tenía ni idea acerca de la programación.

Previo a hacer mi primer programa en lenguaje C, el profesor de ese entonces utilizó una herramienta que, él pensó, podría ayudar con el proceso de aprendizaje, equivocado no estaba. Este apoyo educativo de la cual estoy hablando es Robomind. Esta herramienta me ayudó a comprender los conceptos básicos de programación.

Recuerdo utilizar las instrucciones básicas que hacían que se moviera un robot sobre un tablero, al igual que operaciones aritméticas básicas, la utilización de variables y por muy avanzado, las diferentes estructuras de control. ¿Faltó algo? Sí, y como se retomará más adelante, todas las soluciones en algún punto del curso pueden quedar, por así decirlo, cortas. Por ejemplo, a esta herramienta le hizo falta algo que nos ayudara a mí y a mis compañeros a seguir comprendiendo los últimos dos temas que usualmente se ven en un curso de introducción a la programación, estos son: arreglos y funciones.

Con base a lo anterior, **se busca crear un compilador y un lenguaje de programación propio que sea capaz de ayudar a personas que se encuentran en los niveles superior y media superior a comprender las estructuras básicas de la programación**; sin embargo, para poder ver realizadas las acciones que se generen a partir del lenguaje de programación, me vi en la necesidad de **desarrollar una Máquina Virtual**.

Este trabajo escrito se compone de 2 partes principales, el compilador y la máquina virtual. Durante todo el trabajo escrito se habla de algunas clases esenciales que logran conectar el compilador y la máquina virtual con un IDE (Entorno de Desarrollo). Este entorno solo es para que el usuario pueda interactuar con el compilador a través de la escritura de su programa y visualizar los resultados que entrega la máquina virtual.

La primera parte de este proyecto es crear un compilador. Antes de ese proceso, se detallará **cómo se crea un lenguaje de programación** en el Capítulo 4.

Una vez definido dicho lenguaje, es necesario **construir expresiones regulares** (Capítulo 6.1) que identificarán todos los símbolos del lenguaje y una gramática que ayuda a reconocer si un código fuente está bien escrito o no; al final, se utilizan herramientas de generación de Analizadores Léxicos (JFlex) y Analizadores Sintácticos (CUP).

Al final del desarrollo del compilador, se generará un código objeto, mismo que tendrá que ser interpretado por la máquina virtual (Capítulo 11).

Durante esta primera etapa, se toma en cuenta que el Entorno de Desarrollo de Software que se intenta desarrollar sea atractivo acorde a la edad del público al que se dirige, por

ejemplo, podrá ser no tan llamativo como Scratch u otras soluciones más dirigidas al público infantil.

Las características que contempla el lenguaje de programación que se desarrollará son las siguientes:

- Operadores (Aritméticos, Lógicos, Asignación, Relacionales).
- Tipos de datos (Enteros y Flotantes).
- Soporte de la estructura lógica secuencial.
- Soporte a estructuras de control de selección (if, if else y case).
- Soporte a estructuras de control iterativas (do, do while y for).
- Soporte a funciones (subprocesos).

Terminando la primera parte del desarrollo, se discute sobre los mapas que podrán crearse para el Entorno de Desarrollo. La parte de un mapa es esencial, pues de este mapa dependerá la máquina virtual. El mapa define los límites por dónde el personaje puede moverse en un tablero y los objetos que puede encontrar a su paso. Por consecuencia, en esta parte se define un poco más a detalle sobre cómo estará construido el software integrado.

Al final, se detalla cómo trabajar con el código objeto de un compilador. En esta parte, se desarrollará una máquina virtual que lea el lenguaje objeto de un compilador y muestre resultados casi inmediatos al usuario.

Igualmente, se detalla cada instrucción del lenguaje de programación que se creó y los resultados que se esperan. La máquina virtual es un programa interesante, pues aquí se resuelven todos los componentes de un lenguaje de programación, como pueden ser: resolver operaciones aritméticas, crear y modificar valores de variables, resolver condiciones lógicas y de comparación, resolver los subprogramas que se propusieron, archivos temporales, hilos, y un montón de componentes más.

Esa es la razón por la que este trabajo escrito es grande. Se entrega al final un software que es la integración de un compilador con su máquina virtual que se puede ejecutar en un IDE (Entorno de Desarrollo Integrado).

A grandes rasgos el conjunto de todos los programas que se desarrollarán se visualiza como se muestra en la Figura 1:

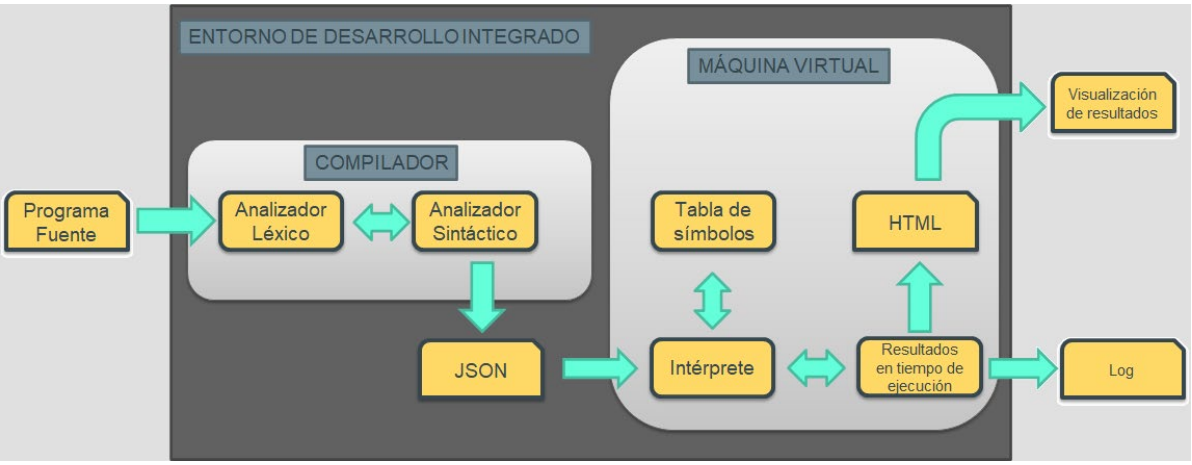

Figura 1 Diagrama del proyecto

1 Teoría Compiladores

El compilador es el componente que será de base para este proyecto. En este capítulo se definirá qué es un lenguaje de programación y los componentes de un compilador.

1.1 Lenguajes de Programación

Para comenzar con la teoría es necesario definir qué es exactamente un lenguaje de programación y cómo se relaciona con los lenguajes naturales, esta palabra será muy utilizada en este trabajo y es muy importante comprender de dónde tiene origen este concepto.

Un lenguaje de programación es parecido al lenguaje natural que usualmente se utiliza en la vida cotidiana. Un lenguaje natural está diseñado para la expresión de ideas y comunicaciones entre personas. Estos lenguajes cubren un amplio espectro de la expresión humana.

Un lenguaje natural es muy parecido a un lenguaje de programación salvo por 2 cosas. La primera es que un lenguaje de programación tiene un dominio expresivo demasiado limitado. Esto debido a que lo que se trata de expresar con los lenguajes de programación son algoritmos, no es un lenguaje que permita expresar ideas como lo solemos hacer en un lenguaje natural. Y la segunda, es que estos solo permitirán la comunicación entre nosotros como personas y los equipos computacionales en el sentido de indicarles a las máquinas las funciones a realizar. Por lo que el diseño de cualquier lenguaje de programación debe satisfacer la comunicación entre una persona y una máquina.

Haciendo un paréntesis, en el párrafo anterior, se mencionó otra palabra clave que es: “algoritmo”. La definición de un algoritmo dice que es una serie de pasos definidos y para el caso de programación, éste determina lo que hace un programa computacional. Las 3 características que definen a un algoritmo, son: que deben ser precisos (tienen un orden específico), definidos (que siempre llega al mismo resultado si se prueba 2 o más veces) y finitos (que tiene un número determinado de pasos). Un ejemplo típico cada vez que se estudia la definición de algoritmo es que se relacione con una receta de cocina. Una receta de cocina tiene una serie de pasos, y por consecuente tiene un principio y un fin. Con un conjunto de entradas se puede llegar a un resultado correcto. Hay que recordar que las computadoras solo siguen instrucciones, en este caso algoritmos.

¿Por qué es importante ver todo lo anterior? Porque existen muchas similitudes entre estos tipos de lenguaje, estas comparaciones son análogas. Para poder diseñar un lenguaje de programación (como se realizará más adelante), es necesario comprender primero el lenguaje natural, ese que se utiliza a diario, y utilizarlo de base para construir uno. Es importante comprender y saber relacionar estos conceptos a la hora de aplicarlos para diseñar un nuevo lenguaje.

1.1.1 Paradigmas de la programación

Como ya se explicó anteriormente, los lenguajes de programación sirven para decirle a una computadora qué hacer a través de un algoritmo.

Estos lenguajes de programación tienen una clasificación, a estas clasificaciones se les conoce como: “Paradigmas de programación”.

Según (A. Tucker & R. Noonan, 2003) estos paradigmas de la programación y sus características son los siguientes:

- Programación Imperativa. Esta clasificación tiene la característica de realizarse en una serie de pasos. Estos pasos realizan cálculos y usualmente tienen entradas y salidas de datos. “Su abstracción son bloques de código, sentencias condicionales, asignaciones, bucles y secuencias”. Algunos ejemplos de lenguajes con estas características son C, C++.
- Programación Orientada a Objetos. Los programas de este paradigma tienen la característica de ser una colección de objetos que interactúan entre sí como pasarse información entre ellos y así modificar su estado. “El modelado, la clasificación y la herencia de objetos son características esenciales”. Algunos ejemplos de lenguajes con estas características son C++, C# y Java,
- Programación Funcional. Los programas que entran en este paradigma tienen la característica de ser funciones matemáticas, con entradas llamadas dominio y resultados llamados intervalos. “Las funciones interactúan y se combinan una con otras utilizando condicionales, recursividad, y composición funcional”. Algunos ejemplos de lenguajes con estas características son Lisp, Scheme, Haskel, F# y ML.
- Programación lógica. Los programas de este paradigma “tienen la característica de ser una colección de declaraciones lógicas sobre el resultado que debería conseguir una función, en lugar de sobre cómo debería conseguirse ese mismo resultado”. Cuando se ejecuta el programa estas declaraciones sirven para obtener una serie de soluciones posibles al problema que se plantea. “La programación lógica ofrece un vehículo natural para la expresión no determinista, que las soluciones a muchos problemas no son únicas sino múltiples”. Algunos ejemplos de lenguajes con estas características son Prolog, Lisp o Erlang.
- Programación guiada por eventos. Este tipo de programación tiene la característica de esperar a un evento (bucle infinito) que no es predecible. Estos eventos pueden ser como acciones del usuario en pantalla (clicks del ratón o de pulsaciones de teclado) o acciones de sensores. Algunos ejemplos de lenguajes con estas características son Visual Basic y Java.
- Programación concurrente. Este tipo de programación son “una colección de procesos cooperativos, que comparten información unos con otros de vez en cuando, pero que operan generalmente de forma asíncrona”. Es decir, son procesos que pueden iniciar en cualquier momento en paralelo con otros procesos sin necesidad de detener procesos. Estos bucles pueden trabajar en paralelo con otros bucles. Algunos ejemplos de lenguajes con estas características son SR, Linda, y Fortran de alto rendimiento.

Un lenguaje puede pertenecer a uno o más paradigmas de programación.

1.1.2 Cualidades de un lenguaje de programación

Para diseñar un lenguaje de programación es necesario tomar en cuenta las siguientes características. Esto con el objetivo de facilitar la interacción entre los programadores y el lenguaje de programación.

Según (A. Tucker & R. Noonan, 2003) estas características se deben tomar en cuenta al momento de diseñar un lenguaje de programación:

1.1.2.1 Simplicidad y claridad

Para cumplir esta característica (A. Tucker & R. Noonan, 2003) sugieren detenerse y pensar: ¿Mi lenguaje será fácil de escribir para alguien que lo utilice? ¿Será fácil de entender para las personas que no realizaron el programa? ¿Es fácil enseñarlo y aprenderlo?

Existen algunos lenguajes muy expresivos como Basic o Pascal, pero hay otros lenguajes que se diseñaron para ser lo menos expresivos como C, aunque con la desventaja de que se vuelven más difíciles de comprender para otra persona. Por ejemplo, en C tomará menos líneas de código para escribir los algoritmos a diferencia de Basic o Pascal.

Un lenguaje debe ser fácil para el programador de escribir, pero no tan difícil de entender. Aunque para esto existen diferentes prácticas de programación.

1.1.2.2 Uniones

Por unión (A. Tucker & R. Noonan, 2003) se refiere a que un elemento de un lenguaje se une a una propiedad en el momento en el que se define dicha propiedad para él.

Es decir, Unir una variable en el momento que se declara, por ejemplo:

int x;

Dónde se une la variable *x* con el tipo *int*. Los principales momentos de la unión según (A. Tucker & R. Noonan, 2003) son las siguientes:

- Momento de definición del lenguaje: Ocurre cuando al momento de definir un lenguaje los tipos de datos se asocian a palabras reservadas que representan. Por ejemplo, en C cuando las palabras reservadas int se relaciona con los números enteros y float con números reales.
- Momento de implementación del lenguaje: Cuando se escribe en un compilador o intérprete, los valores de unen a las representaciones del equipo. Por ejemplo, la representación de un número flotante en memoria.
- Momento de escritura del programa: Sucede cuando se escribe un programa y los nombres de variable se asocian a tipos de datos. Un ejemplo es lo que se hizo más atrás, cuando *x* se asoció con el tipo de dato *int*.
- Momento de la compilación o carga del programa: Cuando se compilan los programas, las variables “estáticas” se asignan a direcciones de memoria fijas, la

pila de tiempo de ejecución se asocia con un bloque de memoria y se asigna el mismo código máquina a otro bloque de memoria.

- Momento de la ejecución del programa: cuando se ejecuta un programa las variables se unen con otros valores, como en la asignación. Por ejemplo, *x = 3.*

Como se pudo observar, un elemento puede unirse a una propiedad en cualquier momento. El concepto de unión temprana significa que un elemento se una a una propiedad tan pronto como sea posible y una unión tardía significa que la unión se postergará hasta el último momento posible.

1.1.2.3 Ortogonalidad

Según (A. Tucker & R. Noonan, 2003) se debe tomar en cuenta lo siguiente: ¿Un símbolo tendrá siempre el mismo significado no importando dónde se utilice? ¿Tiene el lenguaje un pequeño número de características básicas que interactúan de forma predecible, sin importar cómo se combinen en un programa?

Una ortogonalidad se produce cuando se une un nombre en particular con un tipo al escribir un programa y dicha unión no puede cambiar durante el periodo de ejecución. Por ejemplo, asegurarse que una instrucción del lenguaje realice siempre la misma acción. Por otro lado, un contraejemplo de ortogonalidad sería el símbolo “+” en Java, a veces se utiliza cuando se realiza una operación aritmética y en otros se utiliza para concatenación de cadenas.

1.1.2.4 Fiabilidad de los programas

Para poder lograr la fiabilidad de los programas (A. Tucker & R. Noonan, 2003) sugieren tomar en cuenta lo siguiente: ¿El programa se comportará igual cada que se ejecuta con los mismos datos de entrada? ¿Se comporta igual cuando se ejecuta en otras plataformas?

Lo ideal de la fiabilidad es que se pueda ejecutar de la misma manera una y otra vez, así como la compatibilidad de plataformas. Existen algunos programas que restringen los alias y pérdidas de memoria, esto tiene como consecuencia el soporte de tipos estrictos, tienen una semántica, sintaxis bien definidas y soportan la verificación y validación de programas.

1.1.2.5 Aplicabilidad

Para definir la aplicabilidad del lenguaje se puede hacer la siguiente pregunta: ¿El lenguaje diseñado tiene un soporte adecuado para las aplicaciones en las que se utilizará? Por ejemplo, un lenguaje diseñado para la investigación puede no tener utilidad en aplicaciones cliente-servidor.

1.1.2.6 Abstracción

La abstracción es un elemento fundamental del proceso de diseño de un programa según (A. Tucker & R. Noonan, 2003). Un buen programador por lo general busca siempre crear algoritmos para poderlos reutilizar después. Un buen lenguaje de programación soporta

la abstracción de datos y de procesos con el objetivo de ser una herramienta útil en el proceso de programación.

Un ejemplo de abstracción es la utilización de bibliotecas o librerías que realizan una función específica de una manera más sencilla sin tener que volver a escribir desde cero un algoritmo que llegue al mismo resultado.

1.1.2.7 Implementación eficiente

(A. Tucker & R. Noonan, 2003) indican que para lograr una implementación eficiente se tiene que preguntar: ¿permite el lenguaje realizar una implementación práctica y eficiente en las plataformas coetáneas?

Por ejemplo, Algol 68 era un diseño de lenguaje elegante, pero sus especificaciones eran tan complejas que era casi imposible implementarlo de forma efectiva. Las implementaciones de Java igual eran criticadas, sin embargo, con el tiempo se fue mejorando su tiempo de ejecución.

1.2 Mi definición de un lenguaje de programación

Finalmente, con todo lo anterior, se puede definir que un lenguaje de programación es un lenguaje que parte del lenguaje natural para ayudar a modelar algoritmos y así facilitar la comunicación entre personas y máquinas.

Los lenguajes de programación se clasifican en programación imperativa, orientada a objetos, funcional, lógica, guiada por eventos y concurrente. De igual manera, un lenguaje tiene cualidades deseables, aunque no obligatorias, pero de estas características dependerá del diseñador del lenguaje utilizarlos o no.

1.3 Introducción a los compiladores

Todo el software que existe para las computadoras se tuvo que haber escrito en un lenguaje de programación. Sin embargo, antes de que un programa se pueda ejecutar éste debió ser traducido a un formato para que la computadora pueda entenderlo. Aquellos programas que se encargan de traducir se les llama compiladores.

1.3.1 Compiladores

A grandes rasgos un compilador es un programa que recibe un programa fuente y lo transforma en un programa destino. Un compilador ideal debe informar al usuario los errores detectados durante el proceso de traducción.

La figura 2 muestra lo que es un compilador.

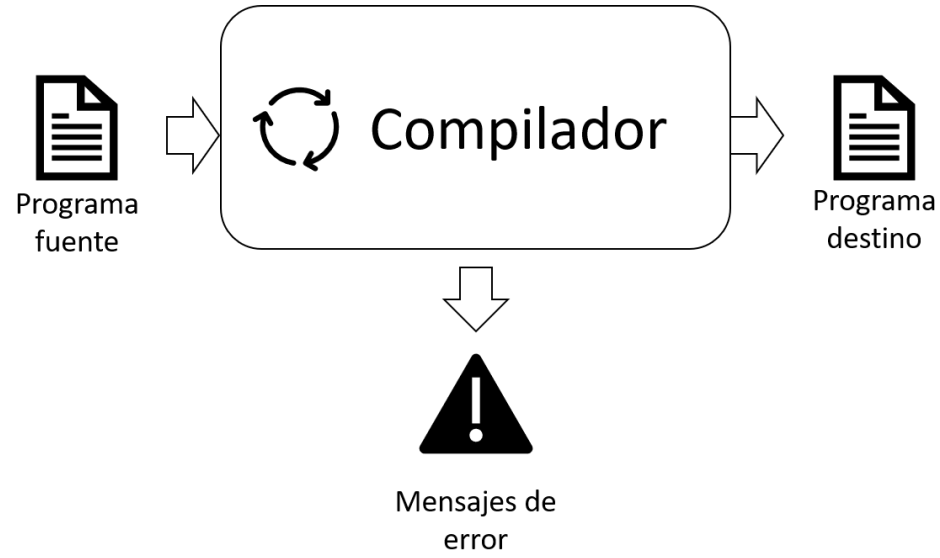

Figura 2 Diagrama de un compilador. Imagen tomada de: (Aho, Lam, Sethi, & Ullman, 2008)

Si un programa destino es un ejecutable en lenguaje máquina, entonces el usuario puede ejecutarlo para producir salidas. Como se muestra en la Figura 3:

Figura 3 Programa Ejecutable. Imagen tomada de: (Aho, Lam, Sethi, & Ullman, 2008)

Un intérprete es otro tipo de procesador del lenguaje, la diferencia es que un intérprete no tendrá un programa destino, sino que podría parecer que ejecuta las instrucciones directamente del programa fuente. Además de que un intérprete es alimentado directamente con datos de entrada.

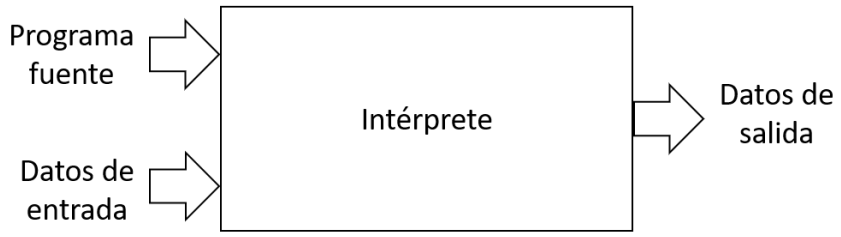

Figura 4 Diagrama de un intérprete. Imagen tomada de: (Aho, Lam, Sethi, & Ullman, 2008)

Para finalizar esta breve introducción, la diferencia entre un compilador y un intérprete se encuentra en el comportamiento del procesador del lenguaje, es decir, si genera o no un programa objeto.

Una ventaja de usar un compilador es que éste sigue una serie de pasos más estrictos. Al final, el compilador resolverá un programa fuente y tendrá como resultado un ejecutable.

Una de las ventajas de usar el intérprete es que, al momento de pasar el procesador del lenguaje éste ejecutará las instrucciones para producir directamente una salida. Y da la sensación de que el intérprete es más rápido y que detectará errores de una forma más rápida.

1.3.2 Estructura de un compilador

Un compilador tiene algunos componentes internos que hacen a este programa interesante.

Hay que imaginar que el compilador es una caja cerrada y se abre, se observará a simple vista que está compuesta por diferentes procesos que se relacionan entre sí.

Lo primero que se podría observar es que inmediatamente después de abrirla sería otra pequeña caja que tiene como objetivo analizar el programa fuente. Esta caja tiene divide el programa en componentes pequeños e impone una estructura gramatical sobre ellas (que es la manera de representar cómo se unen los símbolos para formar palabras y así expresar un significado). Igualmente, esta caja se conecta directamente a otra llamada Tabla de Símbolos de la cual recolecta información sobre lo leído en el programa fuente.

La siguiente caja del circuito sería la caja de la sintaxis. Esta caja recibe información de la Tabla de Símbolos y de la caja de análisis. Tiene como objetivo transformar el programa fuente a otro.

Lo siguiente que se encontrará sería una caja que podría o no aparecer, y es la llamada Optimización del código. Esta tiene como objetivo mejorar el programa destino, esto es opcional, dependerá de cada compilador el utilizarlo o no.

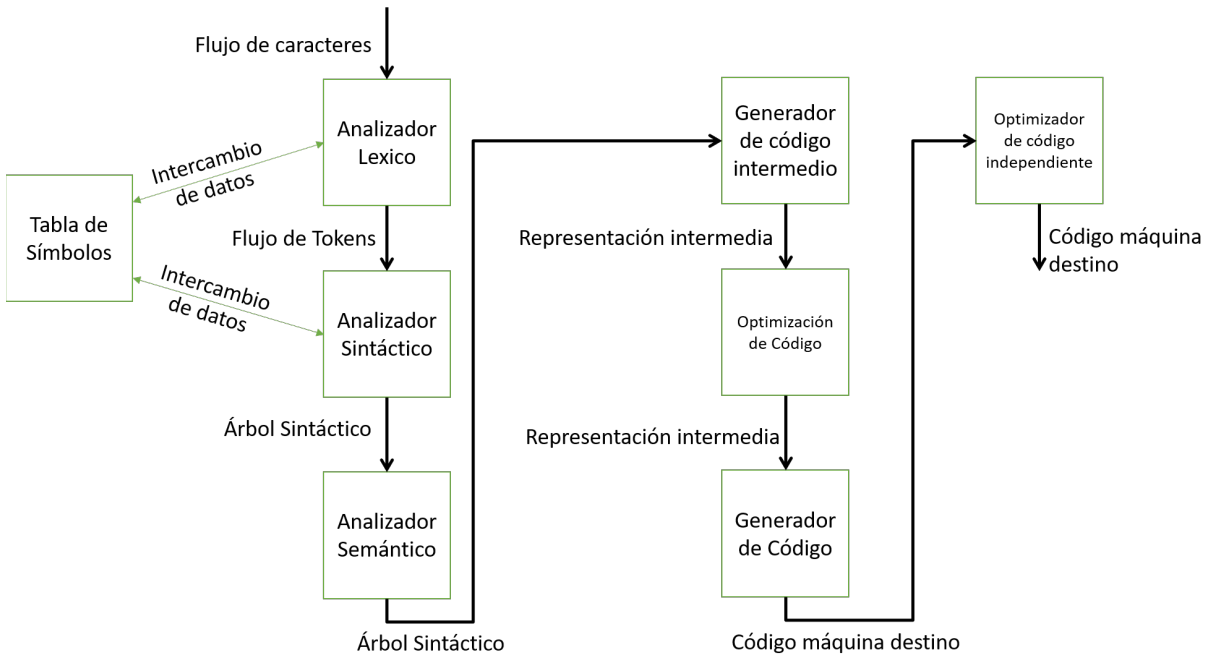

Figura 5 Fases de un compilador. Imagen basada en: (Aho, Lam, Sethi, & Ullman, 2008)

Hasta ahora se conoce qué es un lenguaje de programación, un compilador y la estructura del compilador. A partir del siguiente capítulo se detallará cada uno de los procesos que realiza el compilador.

Es importante ver a detalle cada uno de estos procesos pues el desarrollo de la herramienta integral que acompañará este trabajo escrito retomará cada uno de los capítulos vistos en esta parte teórica.

1.4 Tablas de símbolos

Antes de continuar a los analizadores, es necesario definir qué son las tablas de símbolos.

“Las tablas de símbolos son estructuras de tipos de datos que utilizan los compiladores para guardar información acerca de las construcciones de un programa fuente. La información se recolecta en forma incremental mediante las fases de análisis de un compilador, y las fases de síntesis la utilizan para generar el código destino. Las entradas en la tabla de símbolos contienen información acerca de un identificador, como su cadena de caracteres (o lexema), su tipo, su posición en el espacio de almacenamiento, y cualquier otra información relevante. Por lo general, las tablas de símbolos necesitan soportar varias declaraciones del mismo identificador dentro de un programa.” (Aho, Lam, Sethi, & Ullman, 2008)

1.4.1 La creación de la Tabla de Símbolos

Según (Aho, Lam, Sethi, & Ullman, 2008), los analizadores léxico, sintáctico y semántico son los que crean y utilizan las entradas en la tabla de símbolos durante la fase de análisis.

El Analizador Léxico usualmente es el primero en crear la tabla de símbolos tan pronto como lea un lexema. Ocurre con más frecuencia que el Analizador Léxico sólo puede devolver un token al Analizador Sintáctico, por decir **id**, junto con un apuntador al lexema. El Analizador Sintáctico decide si crea una tabla de símbolos nueva e ir creando entradas nuevas para cada identificador o si debe utilizar la misma que el Analizador Léxico.

1.4.2 La importancia de la Tabla de Símbolos

La tabla de símbolos guarda constantemente información que es útil para el Analizador Léxico y Analizador Sintáctico. La generación de la tabla dependerá de la persona que diseñe el compilador.

Usualmente se crea desde el Analizador Léxico con palabras reservadas e identificadores que vaya detectando en el camino. Para algunos casos puede detectar el valor de los identificadores, pero esto depende de cómo está construido el lenguaje de programación y si hay forma de saberlo desde el análisis léxico.

La tabla de símbolos se utiliza en todo el tiempo si se requiere, ya que es un apoyo de lo ya leído y actualiza nuevos cambios si surgen.

1.5 Analizador Léxico

Repasando el capítulo anterior, un compilador se puede imaginar como una caja que al abrirla se encontrará distintos mecanismos dentro de ella. Uno de los mecanismos es el Analizador Léxico.

El Analizador Léxico será lo primero que se observaría inmediatamente después de ingresar el código fuente. A continuación, se describe con detalle ¿qué es lo que hace realmente?

1.5.1 Lexemas, Tokens y Patrones.

Antes de comenzar, a hablar sobre el Analizador Léxico, creo que es necesario definir 3 palabras que se verán con frecuencia en este capítulo. Estas palabras son términos distintos, pero están relacionados entre sí.

La primera palabra que quiero definir es Token. “El Token es una estructura de 2 valores que consiste en el nombre del token y un valor opcional. El nombre del token es un símbolo abstracto que representa un tipo de unidad léxica; Por ejemplo, una palabra clave específica o una secuencia de caracteres de entrada que denotan a un identificador. “ (Aho, Lam, Sethi, & Ullman, 2008)

La siguiente palabra es patrón. “Un patrón es la descripción de la forma que puede tener un token. Es decir, el patrón es la definición de la secuencia de caracteres que tiene una palabra clave. Se utiliza para identificadores y palabras clave.” (Aho, Lam, Sethi, & Ullman, 2008)

Por último, lexema. “Un lexema es la secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y el Analizador Léxico lo identifica como una instancia de ese objeto.” (Aho, Lam, Sethi, & Ullman, 2008)

1.5.2 La función de un Analizador Léxico

El primer paso de un compilador es leer el programa fuente, cuando está leyendo los caracteres de entrada el Analizador Léxico los agrupará en lexemas para producir de salida una secuencia de tokens.

Cada uno de los Tokens generados son enviados al Analizador Sintáctico para su análisis.

Repasando el capítulo anterior, el Analizador Léxico está conectado con la tabla de símbolos. Y seguidamente interactúa con él. Cada que el Analizador Léxico identifica un lexema que constituye a un identificador debe introducir el lexema a la tabla de símbolos. En varias ocasiones el Analizador Léxico puede leer la tabla de símbolos como ayuda para determinar el token apropiado a enviar al analizador sintáctico.

En la Figura 6 se muestra un diagrama que explica cómo interactúa el Analizador Léxico y Analizador Sintáctico. Se observa que el Analizador Léxico recibe el programa fuente, se encarga de generar los tokens, cuando encuentra uno es enviado al Analizador

Sintáctico. Una vez enviado el Analizador Sintáctico ordena obtener el siguiente token al Analizador Léxico. Estos a su vez, se conectan a la Tabla de símbolos.

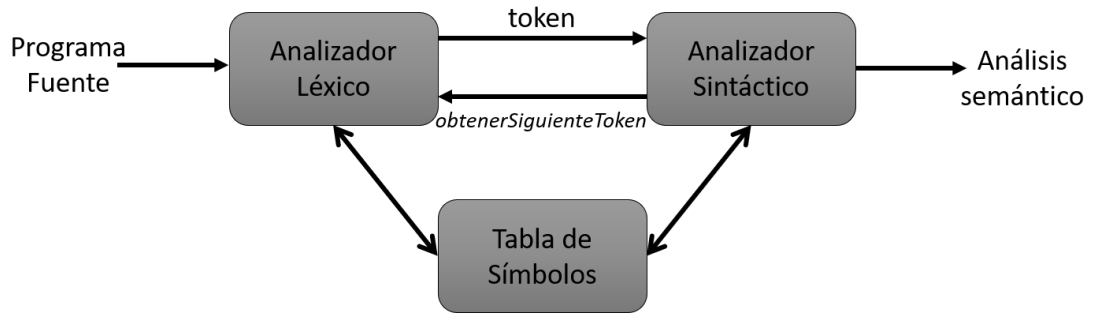

Figura 6 Interacción entre el Analizador Léxico, el Analizador Semántico y la Tabla de Símbolos. Imagen tomada de: (Aho, Lam, Sethi, & Ullman, 2008)

Debido a que el Analizador Léxico "es el primer proceso del compilador que lee el texto de origen (Programa fuente) usualmente realiza otras tareas además de identificar lexemas. Una de esas tareas es eliminar comentarios, espacios en blanco, saltos de líneas, tabuladores o caracteres que ayudan a separar los tokens (Como los puntos y comas). Así mismo, otra de las tareas que realiza será correlacionar los mensajes de error generados por el compilador con el programa fuente." (Aho, Lam, Sethi, & Ullman, 2008) Es decir, el Analizador Léxico puede llevar una cuenta de los caracteres leídos, así como el número de saltos de línea para que en el momento de encontrar un error éste pueda decir exactamente dónde fue encontrado.

En algunas ocasiones los analizadores léxicos se dividen en una cascada de dos procesos:

- El escaneo: "Consiste en un proceso simple que no requiere la determinación de tokens de la entrada, eliminación de comentarios y la compactación de los caracteres de los espacios en blanco consecutivos en uno solo. " (Aho, Lam, Sethi, & Ullman, 2008)
- El análisis léxico: "Consiste en la porción más compleja, en dónde el escanear produce de secuencia de tokens como salida. " (Aho, Lam, Sethi, & Ullman, 2008)

1.5.3 Diferencias entre el Análisis Léxico y el Análisis Sintáctico

Pero ¿por qué es necesario separar los procesos de un análisis léxico con el de un análisis sintáctico? Existen varias razones, las más importantes son las siguientes:

- Una sencillez de diseño. Esta es la más importante de todas. Separar el análisis léxico con el análisis sintáctico permitirá simplificar por lo menos una de estas tareas. Por ejemplo, para un Analizador Sintáctico es más fácil recibir los tokens

ya leídos y filtrados de los espacios en blanco o comentarios. A que el Analizador Sintáctico realice la tarea de limpiar el código. Separar el trabajo entre analizadores simplifica el diseño del lenguaje.

- Se mejora la eficiencia del compilador. Un trabajo del Analizador Léxico permite aplicar tareas de las que solo se especializa el Analizador Léxico. Además, las técnicas de buffer especializadas para leer los caracteres de entrada pueden agilizar la velocidad del compilador en forma considerable.
- Se mejora la portabilidad del compilador. “Las peculiaridades específicas de los dispositivos de entrada pueden restringirse al Analizador Léxico.” (Aho, Lam, Sethi, & Ullman, 2008)

1.5.4 Autómatas Finitos

Un Analizador Léxico parte de la teoría de Autómatas con el objetivo de realizar expresiones regulares. Por esta razón, comenzaré esta parte describiendo cómo es un autómata y cómo se construye uno. La parte formal la detallaré terminando esta breve introducción.

Cuando uno toma un curso de Teoría de la Computación es típico que el primer ejercicio sea imaginarse una máquina que recibe monedas, un cajero automático o una máquina de sodas, eso no importa. Para este ejemplo, hay que imaginar que una máquina que recibirá monedas. Para no hacer la explicación muy larga se limitará a que nuestra máquina recibe monedas de 1 y 2 pesos.

Suponiendo ahora que se comprará un producto cuyo costo será de 3 pesos. Bien se pueden usar 3 monedas de a peso o una moneda de 2 pesos y una de 1 peso. Igualmente, se supondrá que la máquina no dará cambio, por lo que si se ingresan 2 monedas de 2 pesos el producto podrá despacharse.

Bien, ahora se supondrá que se encuentra en una habitación con 2 puertas, no se puede ver más. De pronto, cae una llave para abrir una puerta. Suponiendo que esta llave abre la puerta de 1 peso. Se abre y todo sigue igual, se espera un poco más y vuelve a caer una llave que ahora abre la puerta de 2 pesos. Al entrar a la habitación todo cambia, suponiendo que se puede visualizar la salida y que no se espera ninguna instrucción más.

- La primera habitación cae la primera llave es a la que se llamará a partir de ahora: **estado inicial**. Por ende, cada habitación se tiene que llamar como **estados**.
- Las monedas de 1 peso y 2 pesos son en realidad un conjunto de símbolos, a estos se les llama **símbolos de entrada**.
- Cada que se abre una puerta para pasar a otra habitación en realidad son las **transiciones** de un estado a otro.
- Al final, cuando se visualiza la salida en una habitación y ya no se espera más entradas por recibir, tomará el nombre de **estado final**.

Antes de finalizar esta introducción, se dibujará el autómata.

Primero, se tiene un estado inicial el cual se visualiza como en la Figura 7:

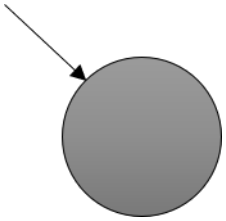

Figura 7 Estado Inicial de un autómata

Hay que recordar que dentro de este estado inicial puede existir 2 caminos, ya sea el 1 o 2. Estos se representan por separado con una transición y estos llevan a un nuevo estado. Cada estado de preferencia deberá ser nombrado, en este caso cada uno de los estados se irán nombrando como {q0, q1, q2, …, qn}.

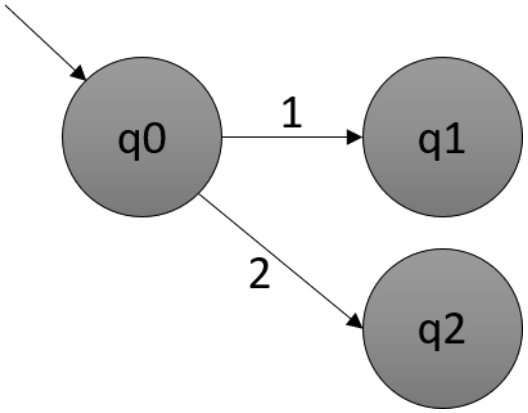

Figura 8 Autómata con transiciones

Para la Figura 8, se tiene que recordar que la primera moneda que se ha insertado fue una moneda de 1 peso, “la puerta” que tiene que seguir es la que está indicada con una flecha y un número 1 llamada **transición**. El camino entonces es el siguiente: inicialmente se está en q0, se pasa por la transición de 1 partiendo de q0 y se tiene que llegar a q1. Así es cómo se representa esta primera transición.

El siguiente paso es tomar a q1 como símbolo inicial, para esto se agregan las 2 transiciones posibles y los nuevos estados con su respectivo nombre. Como sucede en la Figura 9.

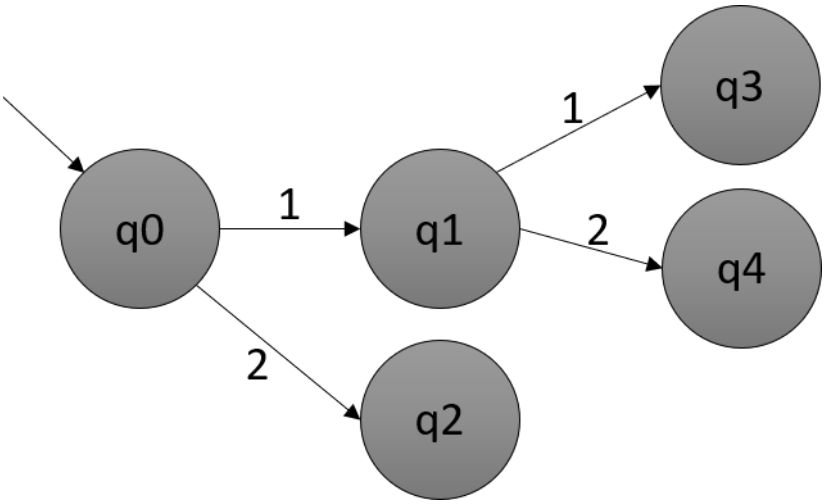

Figura 9 Nuevos estados partiendo de q1

Antes de continuar al siguiente paso, es importante analizar qué sucede en q3. Cuando el autómata ha llegado a q4 debe haber un indicador de que es el paso final, a este estado se le conoce formalmente como “estado final”. Esto se debe a que, si se sigue por el camino de q0 a q1, y de q1 a q4 se llega por así decirlo “a la meta” ya que si se suma el número de las transiciones el resultado es 3, y esto significa que el usuario ha ingresado el monto total de su compra. Usualmente el estado final se representa con un doble círculo. Nótese en la Figura 10 el cambio de q4, este indica que se encuentra en un estado final.

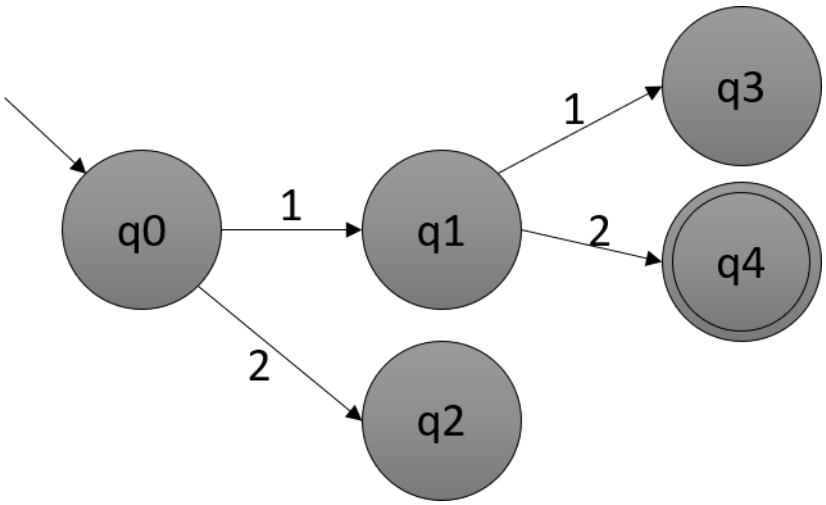

Figura 10 Autómata con estado final

Hasta este punto podría parecer que se ha terminado, pero no es así. Hay que recordar que para esta explicación solo se toma en cuenta lo que pasaría si el usuario ingresa una moneda de 1 peso seguido de una de 2 pesos. Pero el usuario también puede ingresar tres monedas de a peso, o primero una moneda de 2 pesos y luego una de 1 peso, para un caso extremo, nuestra máquina también debería aceptar 2 monedas de 2 pesos porque como se especificó: esta máquina no dará cambio aun cuando el usuario ingrese dinero “de más”.

Continuando con el caso de que el usuario ingresa tres monedas de 1 peso. En el autómata como se muestra en la Figura 09, ya se tiene un “camino” cuando el usuario ingresa 2 monedas seguidas de 1 peso, el camino es el siguiente: de q0 a q1, y de q1 a q3. El siguiente paso es agregar un nuevo estado con su respectiva transición, además, se agrega aquel caso raro dónde el usuario ingresa 2 monedas de 1 peso y al final una moneda de 2 pesos. La última secuencia descrita tiene que ser aceptada. Finalmente, tal y como se observó en la Figura 10, estos estados por agregar serán estados finales. El resultado final se aprecia en la Figura 11:

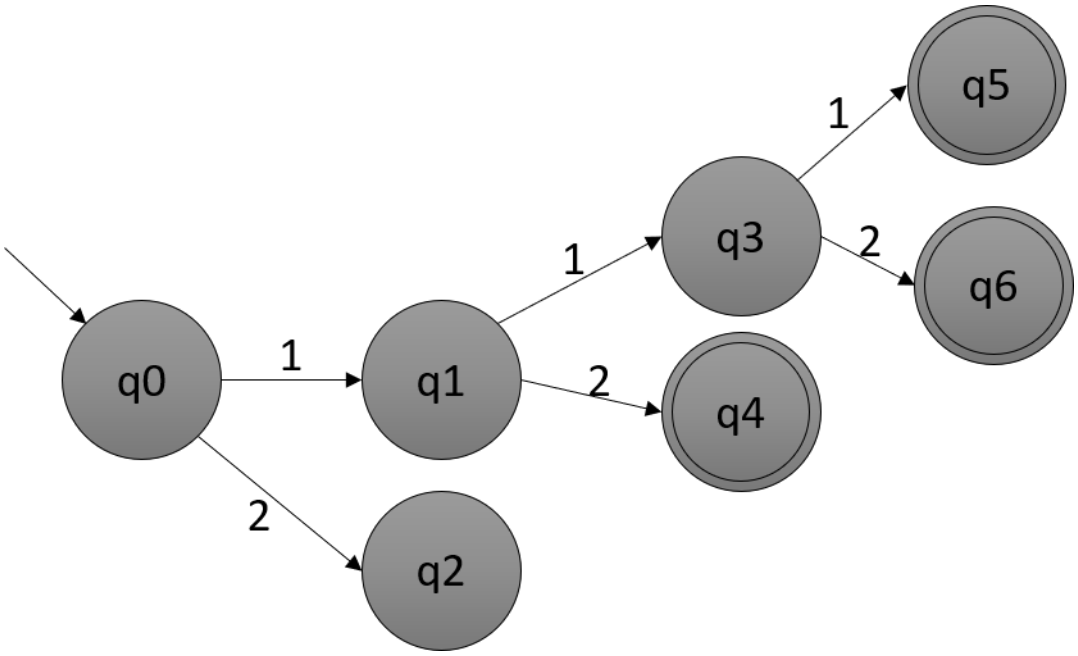

Figura 11 Actualización del Autómata

Finalmente, está el caso del camino de q0 a q2. Al igual que la actualización anterior, se agregarán los estados finales pues los caminos que llevarán a esos estados de aceptación será cuando el usuario ingrese una moneda de 2 pesos seguida de una de 1 peso y 2 monedas de 2 pesos. En la Figura 12 se muestra el autómata final:

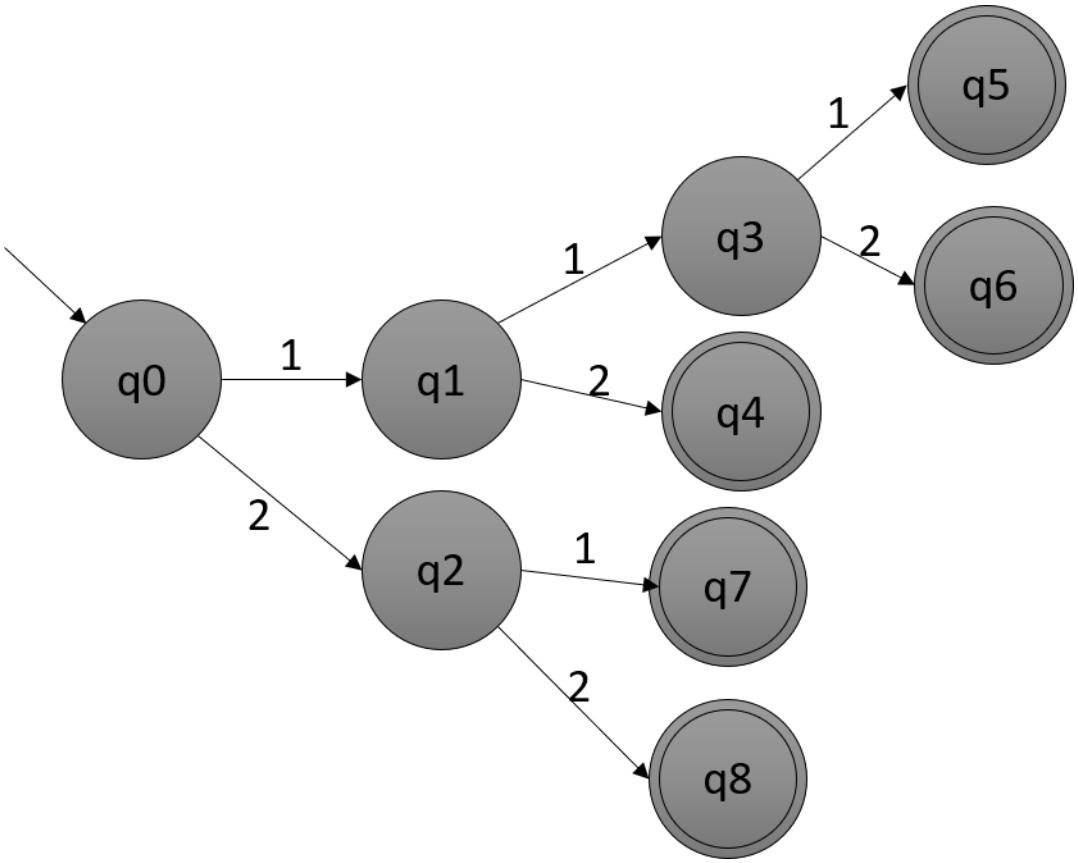

Figura 12 Extensión de q2 del autómata

Por último, hay que tomar en cuenta lo siguiente:

- Sí se pueden reducir los estados finales. Convenientemente se puede ir de q3, y q2 a q4 para así poder reducir los estados y así simplificar el autómata. Sin olvidar colocar sus respectivas transiciones.
- Hablando de transiciones, un truco eficaz para saber si un autómata está bien construido es que los estados tienen la misma cantidad de transiciones que los símbolos de entrada. Salvo que existan restricciones del problema planteado.

1.5.4.1 Definición formal de un Autómata Finito Determinista (AFD)

De acuerdo con (HOPCROFT, MOTWANI, & ULLMAN, 2008) indica que un autómata finito determinista consta de los siguientes elementos:

1. Un conjunto finito de estados, a menudo designado como Q.
2. Un conjunto finito de símbolos de entrada, a menudo designado como Σ.
3. Una función de transición que toma como argumentos un estado y un símbolo de entrada y devuelve un estado. La función de transición se designa habitualmente

como δ. En la representación anterior del autómata, δ se ha representado mediante flechas entre los estados y las etiquetas sobre las flechas. Si q es un estado y a es un símbolo de entrada, entonces δ (q, a) es el estado p tal que existe un arco etiquetado a que va desde q hasta p.

4. Un estado inicial, uno de los estados de Q.
5. Un conjunto de estados finales o de aceptación F. El conjunto F es un subconjunto de Q.

Continúan (HOPCROFT, MOTWANI, & ULLMAN, 2008) explicando que a menudo se hace referencia a un autómata finito determinista mediante su acrónimo: AFD. La representación más sucinta de un AFD consiste en un listado de los cinco componentes anteriores. Normalmente, en las demostraciones, se define como un AFD utilizando la notación de "quíntupla" siguiente:

$$A = (Q, \Sigma, \delta, q0, F)$$

donde A es el nombre del AFD, Q es su conjunto de estados, Σ son los símbolos de entrada, δ es la función de transición, q0 es el estado inicial y F es el conjunto de estados finales.

1.5.4.2 Autómata Finito No Determinista (AFND)

Los autores (HOPCROFT, MOTWANI, & ULLMAN, 2008) igualmente definen las nociones formales asociadas con los autómatas finitos no deterministas e indica las diferencias entre los AFD y AFN. Un AFN se representa esencialmente como un AFD:

$$A = (Q, \Sigma, \delta, q0, F)$$

donde:

1. Q es un conjunto finito de estados.

2. Σ es un conjunto finito de símbolos de entrada.

3. q0, un elemento de Q, es el estado inicial.

4. F, un subconjunto de Q, es el conjunto de estados finales (o de aceptación).

5. δ, la función de transición, es una función que toma como argumentos un estado de Q y un símbolo de entrada de Σ y devuelve un subconjunto de Q.

La única diferencia entre un AFN y un AFD se encuentra en el tipo de valor que devuelve δ: un conjunto de estados en el caso de un AFN y un único estado en el caso de un AFD.

1.5.4.3 Cómo funciona un autómata

Para finalizar, (HOPCROFT, MOTWANI, & ULLMAN, 2008) definen en pocas palabras cómo funciona un autómata:

- "Un AFD tiene un conjunto finito de estados y un conjunto finito de símbolos de entrada".
    - Un estado se tendrá que ser el estado inicial, y cero o más estados tendrán que ser estados de aceptación del autómata.
    - Las transiciones determinan el cambio del cambio de estado en el que se encuentra cada vez que se procesa un símbolo de entrada.
- El autómata acepta cadenas.
    - Una cadena se acepta si al finalizar las transiciones causadas por el procesamiento de símbolos de una cadena, finaliza en un estado de aceptación.
    - Para el caso de un diagrama de transiciones pasará casi lo mismo, si los símbolos leídos forman un camino del estado inicial a un estado de aceptación se considera como válida.

La diferencia entre un AFN y un AFD es que el primero puede llegar a tener cualquier número de transiciones (incluyendo cero) a los estados siguientes para un mismo símbolo de entrada.

1.5.4.4 Para finalizar sobre los autómatas finitos

Si se desea conocer aún más sobre autómatas, existen temas más interesantes sobre ellos tales como:

- Diagramas de transiciones.
- Estados muertos y AFD sin transiciones.
- Autómatas finitos con transiciones épsilon (ε).

1.5.5 Expresiones Regulares

La expresión regular es un tipo de lenguaje que está relacionado con los autómatas finitos no deterministas. Ayudan principalmente para comprender más fácilmente la notación de un autómata.

Las expresiones regulares ofrecen algo que los autómatas no pueden y es "una forma declarativa para expresar las cadenas que se desean aceptar". Como se mencionó en el párrafo anterior, este tipo de lenguaje ayudará como un punto de partida para el Analizador Léxico para que éste pueda procesar el archivo de entrada.

Existen muchas aplicaciones para las expresiones regulares. Entre ellas están la búsqueda en algún archivo en un sistema operativo, la validación de cadenas en un sitio web y por supuesto, para los analizadores léxicos de un compilador.

1.5.5.1 Operadores de una expresión regular

Las siguientes definiciones que fueron tomadas de (HOPCROFT, MOTWANI, & ULLMAN, 2008) ayudan a definir un poco el lenguaje que las expresiones regulares

utilizan. Estas definen las distintas operaciones que se pueden realizar con éstas. Aquí solo hay definiciones, más adelante se utilizarán para formar las expresiones que se requieren para que el Analizador Léxico tome en cuenta al momento de analizar el código fuente.

- La **unión** de dos lenguajes L y M, se representa como L ∪ M, es el conjunto de cadenas que pertenecen a L, a M o a ambos. Por ejemplo, si L = {001, 10, 111} y M = {ε, 001}, entonces L ∪ M = {ε, 10, 001, 111}. Es decir, se unen todas las cadenas de L y M en un mismo conjunto.
- La **concatenación** de los lenguajes L y M son las combinaciones posibles que se pueden formar tomando cualquier cadena de L y concatenándola con cualquier cadena de M. Por ejemplo, si L = {001, 10, 111} y M = {ε, 001}, entonces L·M es {001, 10, 111, 001001, 10001, 111001}.
- La **clausura** (o asterisco, o clausura de Kleene) de un lenguaje L se designa mediante $L*$ y consiste en la posible combinación de cadenas que se pueden formar con los símbolos del conjunto de L, Por ejemplo, si L = {0, 1}, entonces $L*$ es igual a todas las cadenas de 0s y 1s. Si L = {0, 11}, entonces $L*$ constará de aquellas cadenas de 0s y 1s tales que los 1s aparezcan por parejas, como por ejemplo 011, 11110 y ε, pero no 01011 ni 101.

1.5.5.2 Precedencia de los operadores

Al igual que el álgebra que conocemos, existe una precedencia de operadores que aplican a las expresiones regulares. Por ejemplo, para una operación común y corriente como lo es: “xy + z” se sabe que primero se tiene que realizar la multiplicación xy y al final la suma. Y que se realizará lo mismo si se ve la operación de esta forma “(xy)+z”, pero la operación cambia cuando se ve algo así “x(y+z)”. Eso es la precedencia de operadores y a continuación (HOPCROFT, MOTWANI, & ULLMAN, 2008) presentan las reglas para el caso de las expresiones regulares.

1. El operador asterisco (*) es el de precedencia más alta. Es decir, se aplica a los símbolos que aparezcan a la izquierda del asterisco, los símbolos deberán ser el conjunto más corto posible.
2. El siguiente en precedencia es el operador de concatenación, o “punto”. Este operador se aplica después de evaluar el operador (*). Es decir, se evaluarán todas las expresiones junto a las ya evaluadas. La concatenación es asociativa, por lo que no importará el orden en el que se realice. Sin embargo, si se tiene que tomar una decisión se tendrá que aplicar por la izquierda. Por ejemplo, 012 se aplica así: (01)2.

Por último, se aplican todos los operadores de unión (+) a sus operandos. Al igual que el punto anterior, la unión es asociativa y no importa el orden de evaluación, si se requiere se tiene que tomar en cuenta que se calcula por la izquierda. En muchas ocasiones es ideal que la expresión regular que se diseñe no sea evaluada según la precedencia de los operadores. Para eso se utilizan los paréntesis y no está de más utilizarlos, aunque sean previstos por estas reglas.

1.5.5.3 Notación de las expresiones regulares

La siguiente notación se utilizará durante todo el trabajo donde se utilizarán las expresiones regulares. Usualmente es un estándar para las herramientas que se ocuparán, pero aún más que eso en cualquier lado se pueden aplicar. Las reglas son:

1. El símbolo "**.**" (punto) significa "cualquier símbolo".
2. La barra vertical "**|**" expresa una unión. En otras palabras, se usa usualmente para un "OR".
3. El símbolo de interrogación "**?**" significa "cero o uno de lo que está atrás de mí". O sea, que puede estar o no puede estar el símbolo o patrón (Opcionalidad).
4. El asterisco "*****" expresa la cerradura de Kleene. Para propósitos de este trabajo, se usa para decir "lo que está atrás de mí, las veces que sea". Por ejemplo, la siguiente expresión regular significa cualquier palabra que se pueda formar con letras minúsculas: [a-z]*
5. El operador de suma "**+**" significa que se requiere forzosamente más de uno. Es como decir RR*
6. Los corchetes "[" y "]" se usan para delimitar conjuntos de caracteres. También se pueden evitar la barra vertical como, por ejemplo, "[0123456789]".
7. Los paréntesis "(" y ")" se usan para eliminar subexpresiones.
8. La barra de escape "\" ayuda principalmente para secuencias cómo \n, \t o \\ así como representar caracteres individuales como pedir en el lenguaje el símbolo de suma \+ para evitar confundir con los caracteres anteriormente mencionados.
9. El circunflejo "^" expresa complementariedad solo si sigue inmediatamente de un corchete "[", de lo contrario se representa a sí mismo. Por ejemplo, [^0-9] quiere decir: "Que no está entre el 0 y 9".
10. El guion "**-**" indica un rango tras una apertura ("[" o "[^") o antes de un cierre. Por ejemplo: [a-z], o [a-zA-Z0-9]+

1.5.5.4 Las expresiones regulares son autómatas

Los autómatas suelen tener la limitante de no ser lo suficientemente atractivos para representar cadenas de aceptación demasiado complejas. Por ejemplo, un autómata de más de 30 estados, podría resultar algo muy complejo de entender. Afortunadamente existe un equivalente a ellos y son las expresiones regulares.

Las ventajas de las expresiones regulares es que su equivalente algebraico permite representar cadenas de aceptación sencillas y complejas, definen exactamente lo mismo que un autómata, y sobre todo expresar cadenas de aceptación de manera declarativa.

1.5.5.5 Para finalizar con las expresiones regulares

Si se desea conocer aún más sobre expresiones regulares, existen temas interesantes sobre éstas tales como:

- Conversión de un Autómata (AFD) a una Expresión Regular y viceversa.
- Álgebra de expresiones regulares.
- Comprobación de identidades algebraicas.

- Expresiones regulares en UNIX

Reforzando la definición de Expresiones Regulares, es importante recordar que…

- Es una “notación algebraica que describe de forma exacta los mismos lenguajes que los autómatas finitos: los lenguajes regulares” (HOPCROFT, MOTWANI, & ULLMAN, 2008).
- “Los operadores de las expresiones regulares son unión, concatenación (o “punto”) y clausura (o “asterisco”)” (HOPCROFT, MOTWANI, & ULLMAN, 2008).
- Su aplicación ayudará para construir un Analizador Léxico.

1.5.6 ¿Qué es realmente un Analizador Léxico?

Hasta ahora se ha hablado poco del funcionamiento de un Analizador Léxico, y entiendo que hasta este punto el lector se puede hacer la pregunta: ¿Para qué sirven estos conocimientos? Bien, ha llegado el momento de unir cada uno de los bloques que se han visto hasta este momento. Hay que tener presente los conceptos básicos vistos en los capítulos *2.5.2 La función de un Analizador Léxico* y *2.5.4 Autómatas Finitos* para poder formar las expresiones regulares que se pueden utilizar para el compilador.

Primero hay que recordar qué sucede antes de llegar al Analizador Léxico. Regresando a la Figura 01, se tienen como elementos un programa de entrada, un compilador y un programa de salida. Retomando la Figura 04, el primer componente de un compilador inmediatamente ingresado el archivo fuente es el Analizador Léxico.

Lo primero a realizar cuando ingrese un archivo de texto es dividir el programa en Tokens, es importante recordar que estos tienen un nombre y un valor (opcional). Esto se realiza con el objetivo de identificar patrones que son palabras clave. Las palabras clave que identifiquen se convertirán en un lexema, que serán utilizados más adelante por el Analizador Sintáctico.

Estos lexemas son identificados por un autómata finito, que hay que recordar, también son expresiones regulares.

El siguiente ejemplo muestra lo que realiza un Analizador Léxico:

Suponiendo que se tiene el siguiente código fuente, este es un programa sencillo escrito en pseudocódigo:

```
inicio
  entero t
  t = 7
  si t < 5 entonces
    escribir “hey”
  fin si
fin
```

El primer paso del Analizador Léxico es identificar los lexemas basándose en las reglas (expresiones regulares) que Léxico debe seguir. Suponiendo que estas reglas son las siguientes:

Regla	Descripción
inicio|entero|si|entero| entonces|escribir|fin|fin si	Son las palabras reservadas. Una palabra reservada es una palabra que no puede ser utilizada como un identificador.
[a-z]*	Son identificadores. Un identificador es por así decirlo, el nombre de una variable. Ayuda a los programadores a identificar objetos.
=	Operador de asignación. Sirve principalmente para darle valor a un identificador.
<|>	Operadores de comparación. Ayudan al momento de querer realizar una operación lógica.
[“ “|\n]	Blanco o Salto de Línea. Simplemente es un espacio en blanco que ayuda a delimitar el programa.
\” [a-z|” “]* \”	Cadena. Se refiere a identificar una cadena.
[0-9]*	Número. Se refiere a identificar un número.

La tabla anteriormente descrita no es universal, está incompleta y puede ser expandida. Pero para este ejemplo es suficiente identificar solo estos elementos.

El primer paso del analizador es leer símbolo por símbolo (Dividir en tokens) e irlo comparando con sus reglas, por ejemplo, el primer carácter es “i”. Por sí sola “i” puede parecernos un identificador ya que en sus reglas está que un identificador se compone de letras minúsculas de la “a” a la “z”.

El Analizador Léxico continúa leyendo el programa inicial “in”, ahora se están formando patrones, pero aún debe seguir leyendo hasta que se identifique un lexema.

Continúa leyendo hasta formar la palabra “inicio”, seguidamente se encuentra con un salto de línea, con esto, el Analizador Léxico identifica lo ya leído antes del salto de línea y lo compara con sus reglas. Aquí es dónde se identifica que “inicio” es una palabra reservada. Esta palabra leída se vuelve un lexema.

Así seguirá leyendo e identificando cada una de las palabras. Al final, habrá quedado la identificación de lexemas de la siguiente manera:

Lenguaje	Identificación
1. inicio 2. entero t 3. t = 7 4. si t < 5 entonces 5. escribir “hey” 6. fin si 7. fin	1. palabraReservada(inicio) 2. palabraReservada(entero) identificador(t) 3. identificador(t) operadorDeAsignación(=) numero(7) 4. palabraReservada(si) identificador(t) operadorDeComparación(<) palabraReservada(entonces) 5. palabraReservada(Escribir) cadena(hey) 6. palabraReservada(fin si) 7. palabraReservada(fin)

Cada que un lexema es identificado se estará completando la tabla de símbolos, dependerá del desarrollo del compilador qué elementos deben ir en la tabla de símbolos para su posterior consulta.

Recordar que…

- La regla que se toma en cuenta para la identificación de un lexema, es la regla que encaje con la cadena más larga que se pueda leer.
- Los espacios en blanco en muchos casos son irrelevantes y son omitidos, pero siempre deben estar declarados pues de lo contrario arrojaría un error de léxico cada que se encuentre con uno de estos elementos.

Hasta aquí es lo que realiza un Analizador Léxico, en el siguiente capítulo se verá cómo es que se conecta con el Analizador Sintáctico.

1.6 Analizador Sintáctico

Hasta ahora al Analizador Léxico no le interesa saber si lo que está leyendo tiene sentido. Es decir, no conoce reglas para determinar que, por ejemplo, para el lenguaje C se identifica la palabra “main” como una función que se define cómo el punto de partida en cada programa. El Analizador Léxico no le va a importar si se escribe varias veces main como palabra reservada dentro de un mismo archivo, solo se limita a decir: “Está bien escrito y no me importa cómo esté acomodado”. Para esta funcionalidad, están los Analizadores Sintácticos.

Pero antes de continuar, es necesario repasar algunos temas previos a explicar cómo funciona el Analizador Sintáctico y cómo conviven los Analizadores Léxicos y Analizadores Sintácticos.

1.6.1 Gramáticas Libres de Contexto

Acorde a (Aho, Lam, Sethi, & Ullman, 2008), una gramática es la definición de la estructura jerárquica de las instrucciones de un lenguaje de programación. Por ejemplo, para una instrucción if-else su gramática es:

if (expr**)** instr **else** instr

En la gramática anterior, los símbolos en negritas son conocidos como “**terminales**”, y las que no, son conocidos como “**no terminales**”. Los terminales son las palabras que no tienen continuación, estas son palabras claves como “**if**”, los paréntesis “**()**” y la palabra “**else**”. Los no terminales son las palabras que tienen continuación en otra estructura, estas son *expr* e *instr*. La siguiente expresión ejemplifica una segunda estructura para el no terminal *instr*.

instr → **if (**expr**)** instr **else** instr

en dónde la flecha se puede leer como “puede tener la forma”. Para este ejemplo, los elementos de la producción son los mismos a los anteriores.

1.6.1.1 Definición de gramáticas

Los autores (Aho, Lam, Sethi, & Ullman, 2008) explican que una gramática libre de contexto tiene cuatro componentes:

1. Un conjunto de símbolos **terminales**, también llamados como “tokens”. Los terminales son símbolos de gran importancia para una gramática.
2. Un conjunto de **no terminales**, igualmente llamados como “variables sintácticas”. Cada no terminal representa otro conjunto de cadenas o terminales.
3. Un conjunto de **producciones**. Cada producción es un no terminal conocido como “encabezado” (lado izquierdo de la producción), una flecha, y una secuencia de terminales y no terminales llamados “cuerpo” (lado derecho de la producción). El propósito de una producción es especificar de una forma escrita cada instrucción.
4. Se designa un no terminal como **símbolo inicial**.

1.6.1.2 Relación entre los tokens y el Analizador Sintáctico

Hay que recordar que (Aho, Lam, Sethi, & Ullman, 2008) explican que, en un compilador, el Analizador Léxico lee los caracteres del programa fuente, los agrupa en lexemas, y que tiene como salida tokens que representan estos lexemas. Un token usualmente contiene información como el nombre del token y el valor de atributo.

Los nombres del token son símbolos abstractos con los que trabaja el Analizador Sintáctico para su análisis. Estos tokens son llamados terminales en la gramática de un lenguaje de programación.

Si se encuentra presente el valor de atributo, se guarda el valor en la tabla de símbolos que se relaciona con la información del token.

1.6.1.3 Árboles de análisis sintáctico

Acorde a (Aho, Lam, Sethi, & Ullman, 2008), el árbol de análisis sintáctico es la forma gráfica de representar una gramática. Por ejemplo, “Si el no terminal A tiene una producción A → XYZ, entonces un árbol de análisis sintáctico podría tener un nodo interior etiquetado como A, con tres hijos llamados X, Y, y Z de izquierda a derecha”

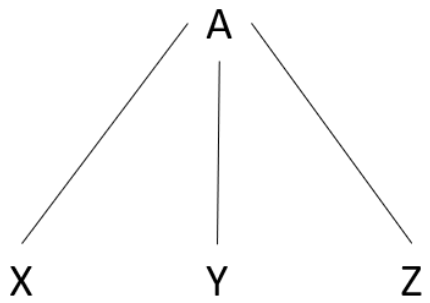

Figura 13 Ejemplo de un árbol sintáctico. Imagen tomada de: (Aho, Lam, Sethi, & Ullman, 2008)

(Aho, Lam, Sethi, & Ullman, 2008) definen que un árbol sintáctico tiene las siguientes propiedades:

1. La raíz de un árbol es el símbolo inicial.
2. Cada hoja terminará siendo un terminal o un símbolo épsilon (ε).
3. Cada nodo interior es un no terminal.
4. Cada rama a partir de un símbolo no terminal deberá tener su respectivo símbolo, sea terminal o no. Esta regla se puede omitir si un símbolo terminal solo contiene un elemento como, por ejemplo, si A → ε es una producción entonces el nodo podrá contener solo un hijo, en este caso el símbolo ε.

1.6.1.4 Análisis Sintáctico LALR

Para propósitos de la herramienta que se utilizará más adelante, será necesario definir qué es una tabla de análisis sintáctico LALR.

Acorde a (Antunez, 2018) un Analizador Sintáctico LALR es de tipo Ascendente, es decir, que construye árboles sintácticos a partir de hojas a la raíz.

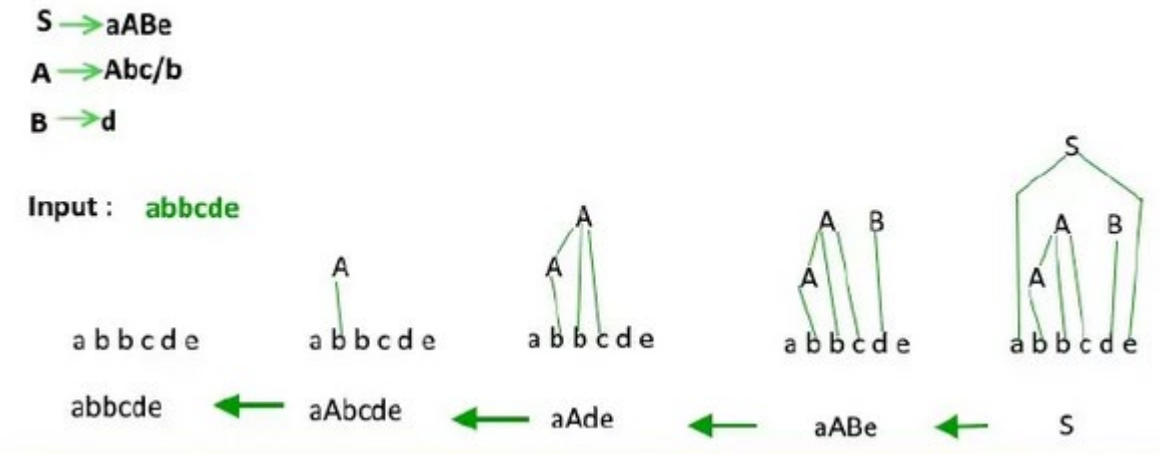

Figura 14 Árbol LALR. Imagen tomada de: (Antunez, 2018)

1.6.1.5 Antes de terminar sobre las gramáticas…

Existen otros temas sobre gramáticas libres de contexto, pero que son interesantes, tales como:

- Ambigüedad.
- Eliminación de ambigüedad.
- Factorización por la izquierda.
- Eliminación de recursividad.
- Gramáticas LL
- Gramáticas LR: SLR Simples
- Tablas de análisis sintáctico SLR
- Tablas de análisis sintáctico LALR

1.6.2 Regresando a los Analizadores Sintácticos

Finalmente, antes de continuar al siguiente capítulo, hay que recordar la función del Analizador Sintáctico.

Hay que recordar el poco el proceso que se lleva hasta ahora. Lo primero que se tiene que hacer es analizar un archivo de texto, este archivo de texto está escrito en un lenguaje de programación. Al principio de este trabajo escrito, se definió qué es un lenguaje de programación. Este tiene ciertas características a cumplir, una de las más importantes es que tiene una estructura que permite principalmente representar comúnmente algoritmos. Para este punto del trabajo importa que tenga una estructura previa.

Lo primero que realiza el compilador es leer y descomponer con un Analizador Léxico el programa fuente en lexemas. Y hasta aquí era lo que se había visto. Pero aquí hay un paso más que realiza el Analizador Léxico y es que se comunica constantemente con el

Analizador Sintáctico. Cada que el Analizador Léxico genera un lexema este lo tiene que enviar al Analizador Sintáctico para su análisis.

El Analizador Sintáctico con base a sus reglas gramaticales analiza lo que está recibiendo, encajando cada lexema y verificando que no exista un error alguno. Si lo hay, se le llama como “error del sintáctico” y debe ser manejado por el usuario final en el front-end (la parte con la que el usuario interactuará con el programa).

El Analizador Sintáctico también consulta comúnmente la tabla de símbolos si lo requiere.

Durante el proceso del Analizador Sintáctico se puede ir generando código intermedio o código destino. Esto dependerá del cómo está diseñado el Analizador Sintáctico.

En la Figura 15 puede observar cómo se lleva a cabo el proceso:

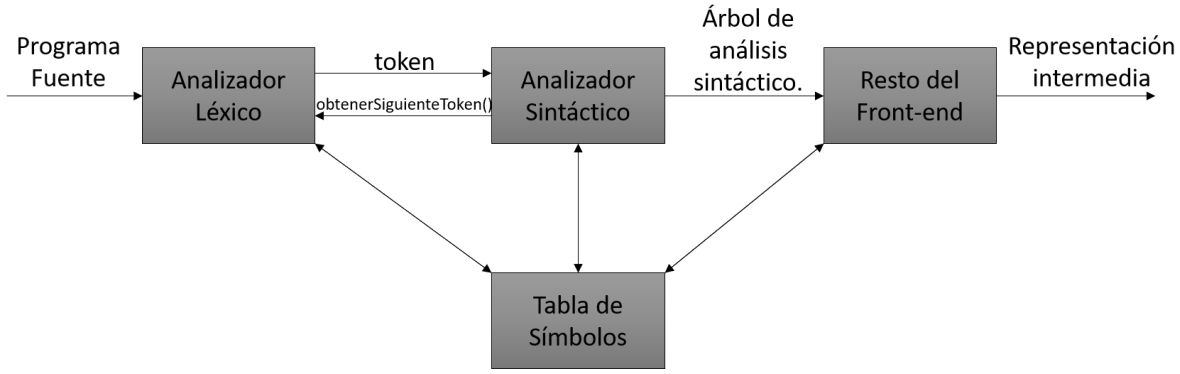

Figura 15 Proceso que ocurre entre analizadores léxicos y sintácticos. Imagen tomada de: (Aho, Lam, Sethi, & Ullman, 2008)

1.7 Otros componentes del compilador

El siguiente bloque son los siguientes pasos que sugieren en la Figura 05, estas definiciones no se utilizarán a lo largo de este proyecto, por lo que se da un breve repaso para no dejarlas sin definir.

1.7.1 Generación de código intermedio

(Aho, Lam, Sethi, & Ullman, 2008) cuentan que durante el proceso de traducir un programa fuente a un código destino puede existir un paso intermedio, esta representación intermedia puede tener diversas formas. Los árboles sintácticos pueden ser una forma de representación intermedia; éstos se utilizan en lo general durante el análisis sintáctico y el análisis semántico.

Una vez que se ha terminado el análisis del programa fuente, muchos compiladores generan un nivel bajo explícito, es decir, una representación intermedia similar al código

máquina, lo anterior, se puede considerar como un programa para una máquina abstracta. Este código intermedio debe tener dos propiedades importantes: Debe ser fácil de producir y fácil de traducir en la máquina destino.

1.7.2 Optimización de código

(Aho, Lam, Sethi, & Ullman, 2008) explican que en esta fase se busca mejorar el código intermedio, de manera de que se produzca un mejor código destino. Generalmente, mejor significa más rápido, pero existen otros objetivos, como pueden ser un código más corto o un código que demanda menos recursos.

1.7.3 Generación de código

Por último, (Aho, Lam, Sethi, & Ullman, 2008) explican que el generador de código recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si en el dado caso de que el lenguaje destino es código máquina, se seleccionan registros o ubicaciones de memoria para las variables que utiliza el programa. Después, estas instrucciones intermedias se traducirán en secuencias de instrucciones de máquina que realizan la misma tarea. Este aspecto importa para la generación de código, puesto que es la asignación jugosa de los registros para guardar variables.

2 Definición de tecnologías

A lo largo de este trabajo se utilizarán diversas tecnologías, una de ellas es JSON. A continuación, se define un poco ¿qué es y para que se utilizan?

2.1 JSON

JavaScript Object Notation (JSON) es un lenguaje muy utilizado por programadores, desarrolladores y profesionales de TI.

Acorde a (Oracle, 2022) este lenguaje es ligero, no requiere de mucha codificación y es rápido de procesar.

2.1.1 Ejemplos y tipos de datos JSON

Una de las mejores características de JSON es que es muy fácil de leer y escribir para los programadores como indica (Oracle, 2022). Igualmente, permite que cualquier software lo analice y genere con facilidad. Se utiliza usualmente para serializar datos estructurados e intercambiar datos entre un servidor y aplicaciones web, pero se podría utilizar en otros proyectos.

JSON está formado por tipos de datos.

- Cadena
- Número
- Booleano
- Nulo
- Almacenamiento
- Matriz

2.1.2 Principales casos de uso para JSON

Estos son los principales usos en dónde puede ser utilizado JSON

1. *Generación de un objeto JSON a partir de datos generados por el usuario*. JSON es perfecto para el almacenamiento de datos temporales. Por ejemplo, en el envío de datos entre aplicaciones web o envío de datos entre programas de computación.
2. *Transferencia de datos entre sistemas.* Como se indicó anteriormente, JSON puede ser utilizado para aplicaciones web para intercambiar datos.
3. *Configuración de datos para aplicaciones.* Al desarrollar aplicaciones, puede requerirse que cada aplicación guarde datos como la dirección IP de un servidor entre otras cosas. JSON puede ser utilizado para este propósito garantizando la disponibilidad de los datos.
4. *Simplificación de modelos de datos complejos.* JSON puede simplificar documentos complejos en componentes más pequeños con datos concretos. Creando así, datos más predecibles y legibles para un humano.

Estas ventajas de JSON son las que permitirán después, avanzar con el proyecto. Pero aún no adelanto nada.

2.2 Máquinas Virtuales

Según (Feng Li , 2017) las máquinas virtuales han existido desde hace muchas décadas en varias formas. Pero fue fuertemente popularizada en 1995 cuando Sun Microsystem publicó su lenguaje de programación Java que utilizaba la Máquina Virtual de Java (JVM).

2.2.1 Tipos de máquinas virtuales

(Feng Li , 2017) menciona que las máquinas virtuales son un sistema informático. El objetivo de estos sistemas es ejecutar lógicas programadas. Las lógicas pueden ser de un nivel muy bajo o un nivel alto. Según el autor, las máquinas virtuales pueden clasificarse en cuatro tipos los cuales son:

1) Máquinas Virtuales de conjunto e instrucciones (Full instruction set architecture - ISA) Proporciona una emulación de una máquina virtual que se puede ejecutar en un ordenador. Por ejemplo, VirtualBox, QEMU o XEN.
2) Las máquinas virtuales de interfaz binaria (Application Binary Interface - ABI) proporcionan una emulación ABI. Es decir, las aplicaciones pueden correr procesos lado a lado con otros procesos de aplicaciones nativas. Un ejemplo de estas máquinas virtuales podría ser la máquina virtual Rosetta de Apple, que se utiliza para la emulación de instrucciones de aplicaciones hechas para otro tipo de procesadores.
3) Máquinas virtuales ISA. proporcionan un motor de tiempo de ejecución para que las aplicaciones que fueron codificadas para ejecutarse en ella lo puedan hacer. Esta máquina se define como un alto nivel y de alcance limitado de la semántica ISA, por lo que no se requiere que la máquina virtual emule un sistema completo. Ejemplos de esta máquina virtual son: JVM (Java Virtual Machine), Microsoft Common Lenguage Runtime o Parrot virtual machine.
4) Language virtual machine (Máquina virtual de lenguaje) proporciona un motor de ejecución que ejecuta programas expresados en un lenguaje huésped. Los programas de estas máquinas virtuales tienen forma de código fuente que no han sido compilados en código máquina. El motor de ejecución necesita interpretar o traducir el programa, así como gestionar la memoria. Por ejemplo, los motores de ejecución para Basic, Lisp, Tcl y Ruby.

Clasificando las máquinas virtuales anteriormente descritas podrían quedar de la siguiente forma:

Las primeras 2 tendrían la característica de ejecutar sistemas operativos o aplicaciones que fueron desarrolladas para ISA o ABI distintas a la del sistema operativo anfitrión. Son comúnmente llamadas “emuladores”.

Las otras 2 máquinas son motores de ejecución cuyo objetivo principal es ejecutar lógicas programadas en forma de ISA virtual o lenguaje huésped.

2.2.2 ¿Por qué una máquina virtual?

Las máquinas virtuales son indispensables para la programación moderna, continúa (Feng Li , 2017). Estas ayudan enormemente a la seguridad informática, productividad de la programación, y portabilidad de las aplicaciones.

Igualmente, son necesarias para lenguajes seguros, esto significa que, lenguajes que se ejecuten en máquinas virtuales tienen la principal característica de seguridad en memoria, seguridad en operación y seguridad de control. Gracias a esto, es más fácil detectar errores de programa o de ejecución de forma temprana y segura.

Estos tres candados de seguridad anteriormente descritas, significan lo siguiente:

- La seguridad de memoria garantiza que un determinado tipo de datos en memoria siempre siga con restricciones. Por ejemplo, una variable de tipo puntero nunca apunta una locación ilegal o un arreglo que no sobrepase sus límites.
- La seguridad de las operaciones, asegura que estas sigan las restricciones de ese tipo. Por ejemplo, una variable de tipo puntero, no permite operaciones arbitrarias sobre ella.
- La seguridad de control garantiza que el flujo de ejecución del código nunca se llegue a un punto o se desboque, por ejemplo, saltar a un bloque a un código malicioso.

La mayoría de lenguajes modernos son lenguajes seguros, aunque sus grados de seguridad pueden ser diferentes.

En el soporte de lenguajes seguros, es necesaria una máquina virtual porque un lenguaje por sí mismo no puede soportar características de seguridad. Por ejemplo, el programa no debe asignar un trozo de memoria que no tiene un tipo asociado, y requiere de una máquina virtual para que le proporcione la memoria para ese propósito, como un determinado tipo de objeto.

Una máquina virtual ayuda a la portabilidad pues las aplicaciones pueden ejecutarse en cualquier sistema operativo que tenga instalada la máquina virtual.

3 Presentación del proyecto

Bien, una vez que se ha visto la parte teórica, ha llegado el momento de hablar del proyecto en general. Si tuviera que resumirlo en un pequeño párrafo diría que se busca **desarrollar una herramienta que sea de apoyo para que una persona mayor a 15 años aprenda a programar**.

La explicación anterior es, en general, lo que se busca realizar y lo que terminará siendo.

3.1 Estado del arte

Esto lleva a investigar un poco. ¿Qué es lo que existe actualmente en el mercado en herramientas para aprender a programar?

3.1.1 Robomind

Robomind es un proyecto creado en 2005 por un estudiante de la Universidad de Ámsterdam. Su uso es principalmente utilizado por escuelas primarias. Soporta algunas estructuras básicas para moverse en un tablero que puede ser creado por cualquier persona. Así como estructuras de programación, tales como: Comentarios, bucles, estructuras de control, funciones y una forma para detener la ejecución del programa cuando se ejecuta. De las cosas más interesantes de este programa es el soporte para Lego Mindstorms.

Anteriormente fue un software de paga, y restringía algunas funcionalidades (Así lo era cuando yo aprendí a usarlo). Ha cambiado demasiado desde que lo usé. Hoy en día la herramienta es gratuita, aunque actualmente están más enfocados a vender cursos de programación y en el mantenimiento de la versión web de Robomind.

Entre algunos defectos sobre su lenguaje de programación (Llamado ROBO) que encuentro son que no existe algo parecido a lenguaje C que es una función principal (Hablo de la función main), o las instrucciones jamás llevarán un delimitador como el punto y coma.

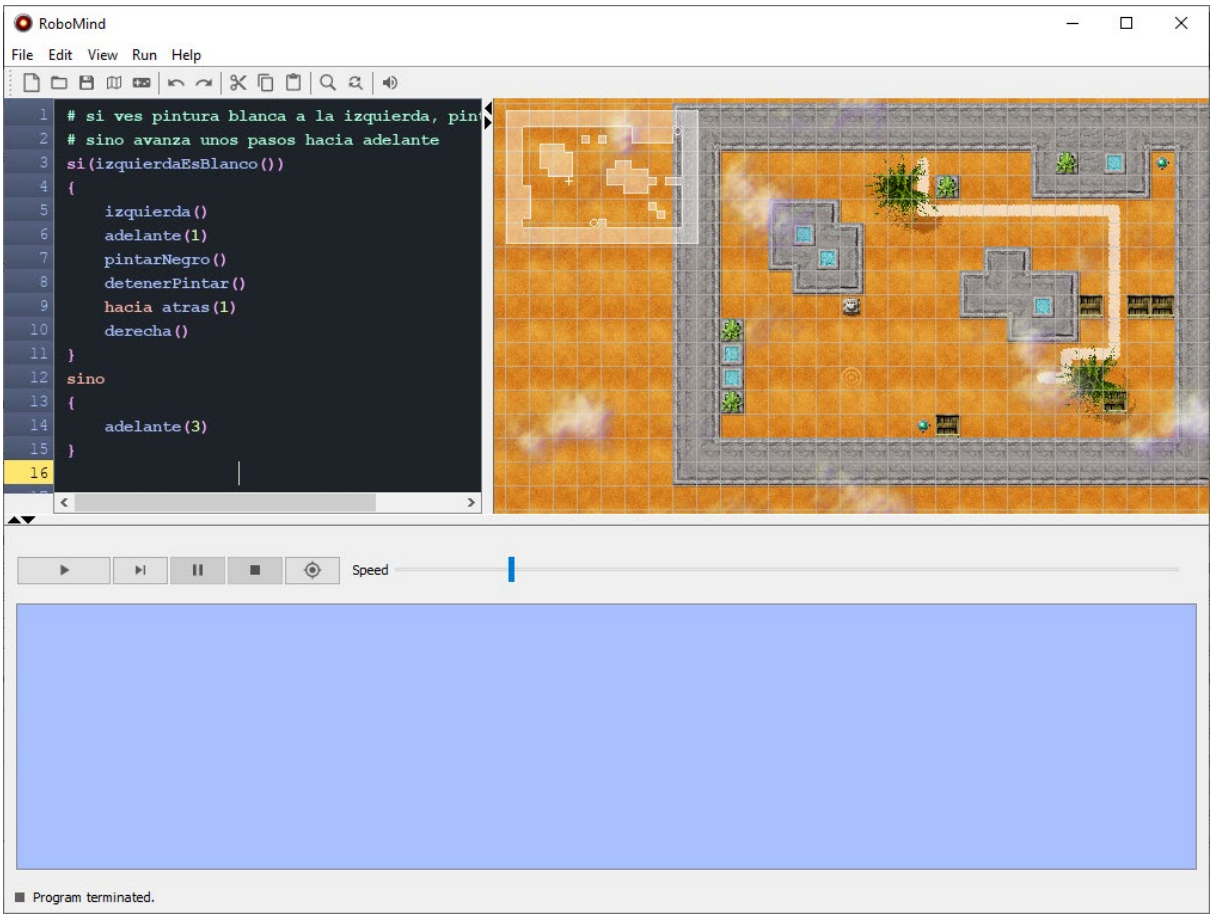

Figura 16 Captura de pantalla del programa Robomind

3.1.2 Lego Mindstorms

Debido a que lo mencioné en la herramienta pasada es momento de hablar de Lego Mindstorms. Este es el más apantallante de todas las herramientas que se presentan.

Esta herramienta va más dirigida a personas que quieren aprender sobre robótica, pues gran parte del tiempo dedicas a crear diseños para lo que se requiera. Consiste de un aparato llamado bloque que se debe programar para realizar acciones. Estas acciones controlan sensores y motores para que un armado pueda moverse.

Como se observó anteriormente, hay un bastante número de herramientas no oficiales que permiten interactuar con estos bloques para programarlos. Su lenguaje de programación no está del todo claro, pues como dije, cada herramienta tiene lo suyo. Pero analizando la herramienta oficial, no hay lenguaje. Al igual que una herramienta de la que se hablará más adelante, su entorno de lenguaje está basado en bloques gráficos, es decir, nunca se escribirá código. Es un método basado en arrastrar y soltar para configurar.

Las desventajas de utilizar Lego Mindstorms para aprender a programar, son varias. La primera es que se dedica gran parte del tiempo en armar con legos una pieza que se pueda utilizar. Ni se hable del tiempo que se llevaría diseñar algo nuevo, en la gran mayoría de casos no se diseña algo nuevo, sino que se siguen instrucciones, por ejemplo, instrucciones de un foro. Segundo, la herramienta oficial es un ambiente gráfico, es decir, no se escribe código, sino que está basado en arrastrar y soltar bloques. Tercero, por desgracia no es accesible para todos, la última versión tiene un costo aproximado de $8000 pesos mexicanos.

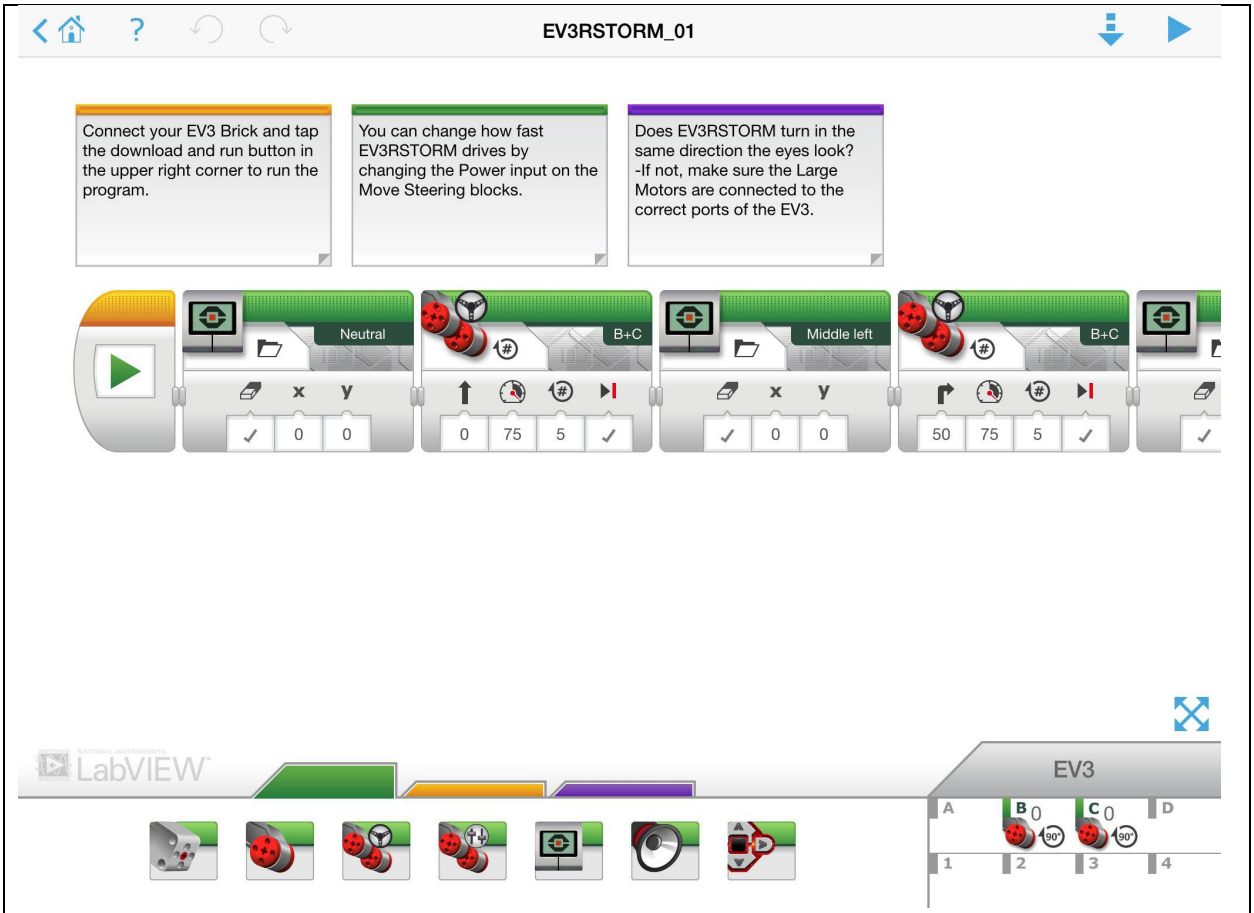

Figura 17 Programa oficial para programar el aparato de Lego Mindstorms. Imagen tomada de: (APKPure, 2019)

3.1.3 Greenfoot

Greenfoot es otra herramienta que se puede utilizar para este propósito, pero va más orientada a la programación orientada a objetos. Sirve más para crear pequeños videojuegos, y su lenguaje de programación, como tal, es Java.

Es decir, que para crear esos pequeños juegos se debe conocer un poco de Java. No es una herramienta tan enfocada a aprender las bases de programación.

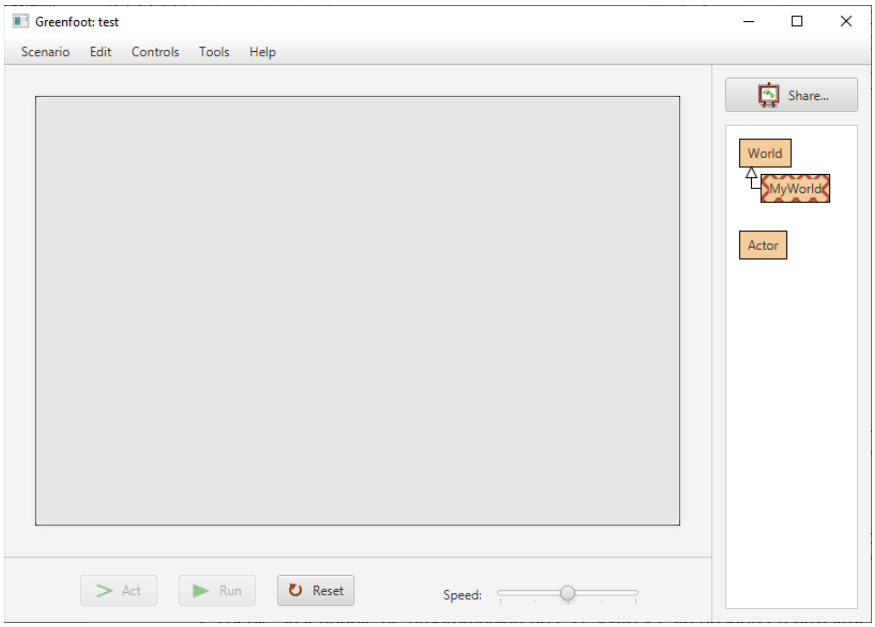

Figura 18 Entorno principal de Greenfoot

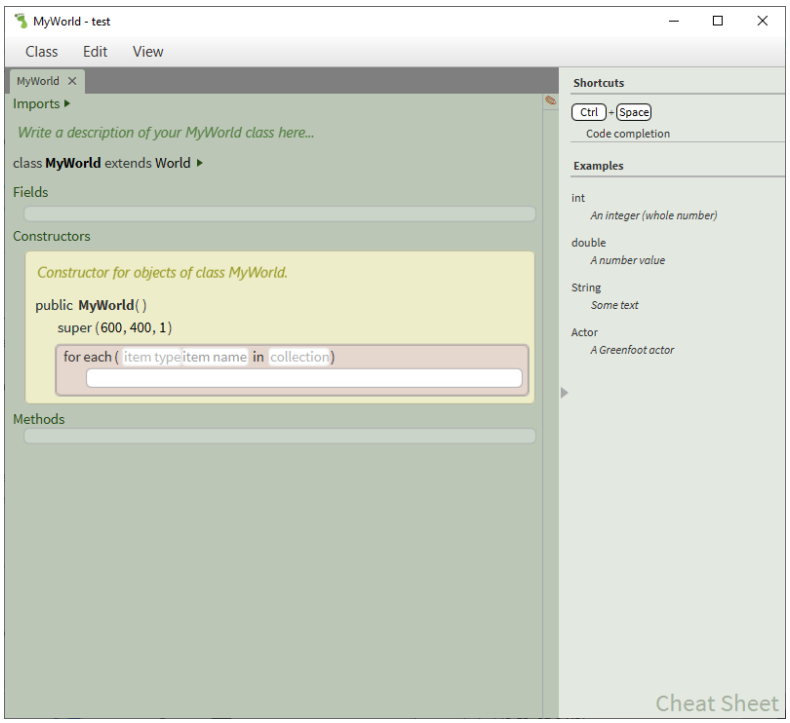

Figura 19 Entorno de programación de Greenfoot

3.1.4 Scratch

Lo anterior me hizo recordar a la herramienta Scratch. Al igual que Greenfoot, permite crear pequeños videojuegos con la diferencia que para programarlos se requiere acomodar bloques parecidos a los que se encuentran en Lego Mindstorms.

Es decir, su lenguaje de programación no es escrito y está basado en arrastrar, soltar y configurar.

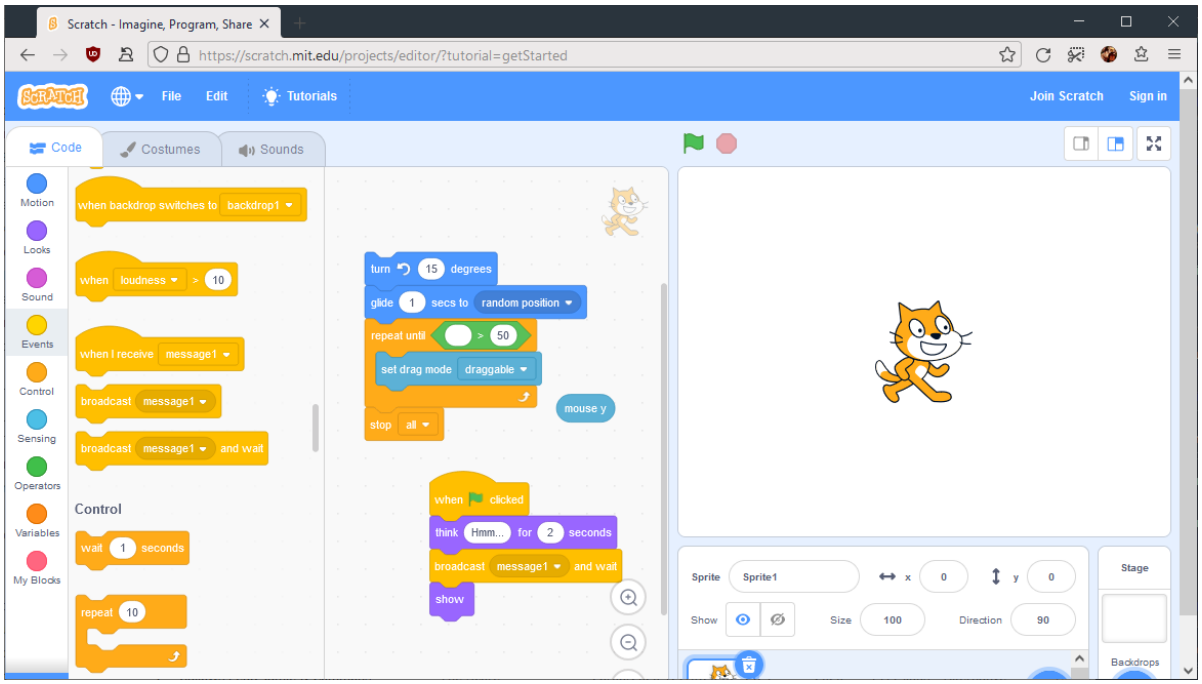

Figura 20 Entorno principal de trabajo de Scratch Web

3.1.5 Karel

Karel es por así decirlo, un compilador que puede adaptarse a distintos proyectos. De Karel se va a encontrar diferentes entornos de desarrollo, pero un mismo compilador.

La creación de mapas para Karel consta de 2 objetos: paredes y de objetos con los que se puede interactuar. Su lenguaje de programación es básico, incluye instrucciones que deben estar dentro del programa principal, y tampoco incluye alguna instrucción dónde utilicen arreglos.

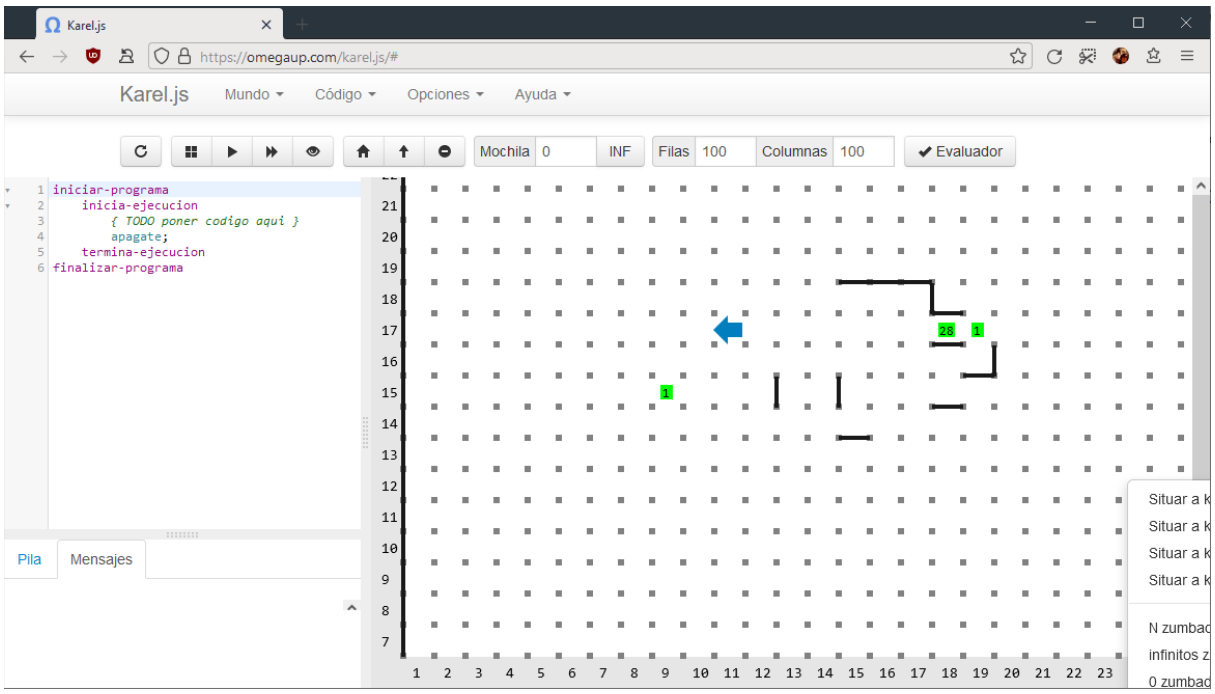

Figura 21 Entorno de desarrollo de una versión de Karel Web

3.1.6 Alice

Alice puede considerarse una versión más sencilla que Greenfoot. Su lenguaje de programación es igual a algunas soluciones anteriormente mostradas, es de estilo arrastrar, soltar y configurar.

Al igual que soluciones anteriores, este entorno está diseñado para crear pequeños videojuegos.

Puede ser utilizado para hacer una introducción a la programación orientada a objetos e introducción a la programación por eventos.

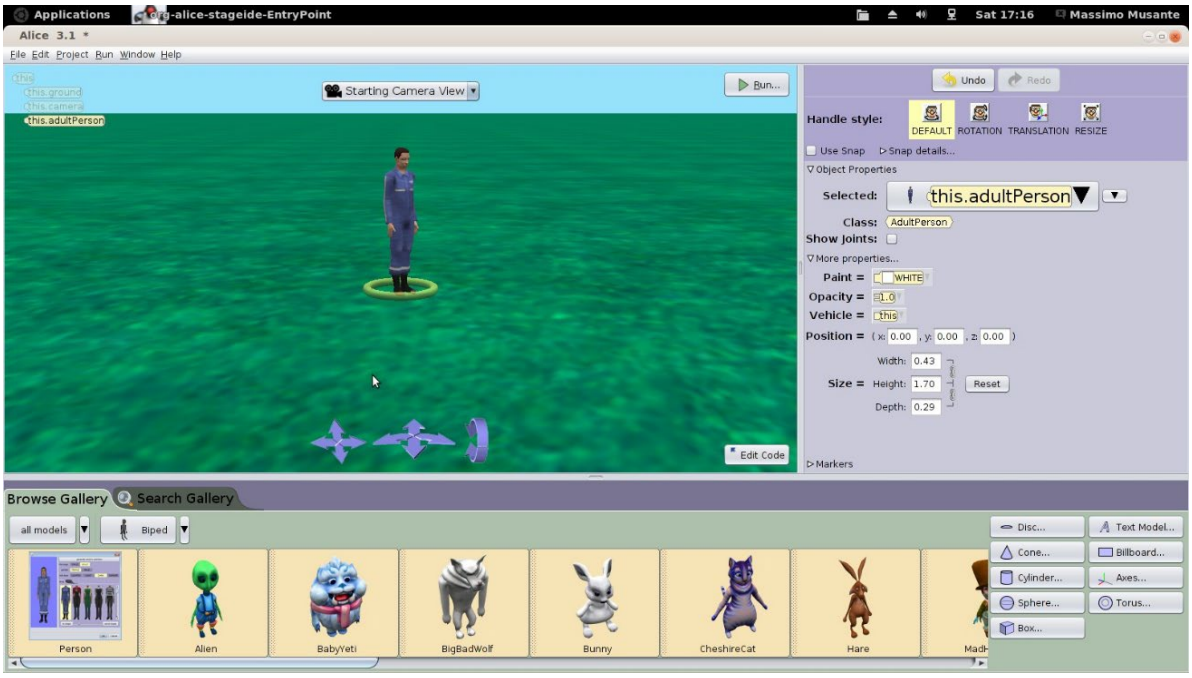

Figura 22 Entorno de trabajo de Alice. Imagen tomada de: (massimomusante, 2012)

3.1.7 Kodu

Kodu es un videojuego que igualmente fue creado con el propósito de ser usado como herramienta para enseñar a programar.

Se centra mayormente a la programación por eventos, y sus características están en crear mundos para compartirlos con el mundo.

Su lenguaje de programación es parecido a lo que ya se observó anteriormente, que es, arrastrar, soltar y configurar.

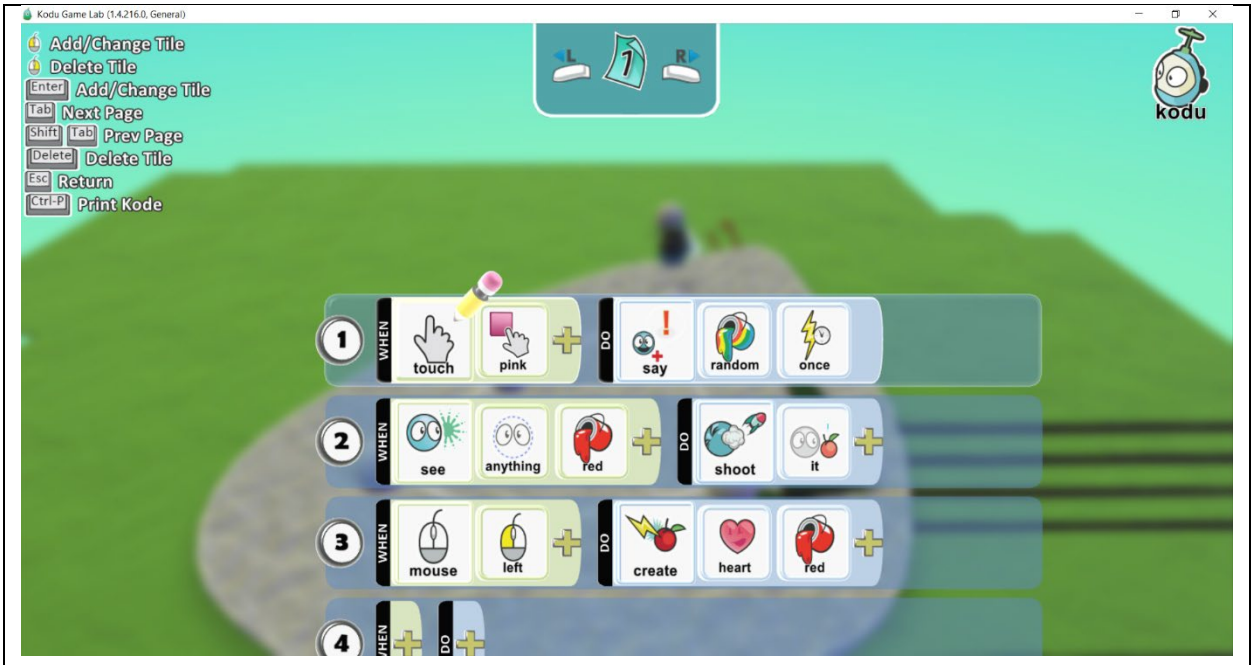

Figura 23 Entorno de trabajo de Kodu. Imagen tomada de: (Matte, 2018)

3.2 Más información del proyecto

Después de analizar las herramientas anteriores, se proponen las siguientes características que podría tener el sistema de software final:

- Se intentará crear **un programa capaz de ayudar en el aprendizaje** para las personas que se encuentren preferentemente en los niveles medio superior o superior.
- Se intentará **crear un lenguaje de programación** que esté al nivel del público objetivo.
- El paradigma de programación al que se enfocará este proyecto será la **programación imperativa**.
- El objetivo de este proyecto es **ayudar a estas personas que van iniciando a programar, facilitar la comprensión de las estructuras básicas** antes de aprender un lenguaje de programación de alto nivel.

- Debido al público al que va dirigido, se busca que el programa sea atractivo acorde a la edad del público al que se dirige, por ejemplo, podrá ser no tan llamativo como Scratch u otras soluciones más dirigidas al público infantil que se encuentran en el mercado.

Lo anterior es básicamente lo que terminará siendo el proyecto. Pero aún se tiene que definir las características del lenguaje de programación. A continuación, se enlistarán los temas que debe cubrir el lenguaje de programación a crear:

- Operadores (Aritméticos, Lógicos, Asignación, Relacionales).
- Tipos de datos (Enteros y Flotantes).
- Soporte de la estructura lógica secuencial.
- Soporte a estructuras de control de selección (if, if else y case).
- Soporte a estructuras de control iterativas (do, do while y for).
- Soporte de Arreglos unidimensionales.
- Soporte de Arreglos bidireccionales.
- Soporte a funciones (subprocesos).

La razón de la lista anterior es cubrir los temas que **usualmente se ven en un curso introductorio de programación**. En la siguiente lista se definen los limitantes del proyecto:

- Se busca desarrollar una aplicación de escritorio, esto para que pueda ser utilizado en ámbitos educativos. Para enseñar a programar se requiere de una computadora.
- Sobre los requerimientos del software se busca que se pueda ejecutar en cualquier equipo capaz de soportar la máquina virtual de java. Cualquier computadora moderna no debería tener problema.
- No se contempla que sea web. Pero se diseñará de tal forma que en algún futuro algún otro estudiante pueda pasar este proyecto a un sitio web como lo hace Karel.
- El programa estará disponible en dos idiomas: inglés y español.

¿Qué hay de algunas extensiones futuras?

- Se buscará que en un futuro pueda ser utilizado para comprender las estructuras de datos (Pilas, colas, listas y árboles) y/o la Programación Orientada a Objetos. Sin embargo, no está siendo contemplado de momento el desarrollo de lo anteriormente mencionado.

Con base en lo anterior, se construirán algunos bocetos de lo que podría ser el programa final. Estos bocetos son una etapa temprana de lo que podría llegar a ser el programa final. En Ingeniería de Software a esta etapa del proceso se le conoce como: Análisis.

3.3 ¿Qué es lo que se desarrollará?

El concepto del producto final es el siguiente:

Me estoy basando fuertemente en lo que es Robomind y Karel. En ambas herramientas **un robot se mueve sobre un tablero** con instrucciones, estas instrucciones pueden ser mover, o pintar en el suelo.

A grandes rasgos, este programa constará de distintos programas que trabajarán en conjunto. Las cuales son las siguientes:

Para crear diversos tableros de juego se tendrá un programa que analice un lenguaje de tipo declarativo, es decir, no se requerirá de un compilador para mostrar su contenido. El tablero deberá contener dónde inicia el personaje, qué obstáculos tiene el mapa, o una meta a dónde llegar. Más adelante se detallará más al respecto.

El siguiente programa será el compilador. Este será el encargado de leer el código fuente del usuario para transformarlo en un código destino.

Una vez que se tiene el código destino es necesario analizar los resultados. Esto se logra con la máquina virtual.

La última capa es la interfaz gráfica con la que interactúa el usuario. En esta interfaz el usuario introduce su programa, controla lo que está realizando otros componentes como la máquina virtual (detenerla o ejecutarla) y dónde se muestran los resultados de una forma amigable.

Finalmente, el caso de uso del proyecto será el siguiente:

1. El usuario abre el programa.
2. El usuario carga un mapa que fue realizado por alguien más o trabaja sobre el predeterminado.
3. El usuario se pone a programar solucionando el problema encargado.
4. El usuario ejecuta su programa.
5. El programa lee el código destino y comienza a mostrar los resultados moviendo un personaje sobre el tablero.
6. Se repite el paso 3 a 5 las veces que sea necesario.

Por personaje me refiero al equivalente que es el robot en Robomind o Karel. A lo largo de este trabajo escrito, se referirá como Personaje, debido a que aún no se define si este personaje es un robot o no.

3.4 El Boceto

La pantalla principal del programa tendrá un espacio de editor de código y un espacio para visualizar un mapa.

El editor de código servirá para introducir el código del lenguaje de programación que creará.

El espacio del mapa permitirá visualizar el mapa con algún tipo de problema a solucionar.

Figura 24 Boceto del espacio de trabajo

En la Figura 25 se muestra un pequeño boceto de cómo quedaría formado el espacio de trabajo principal:

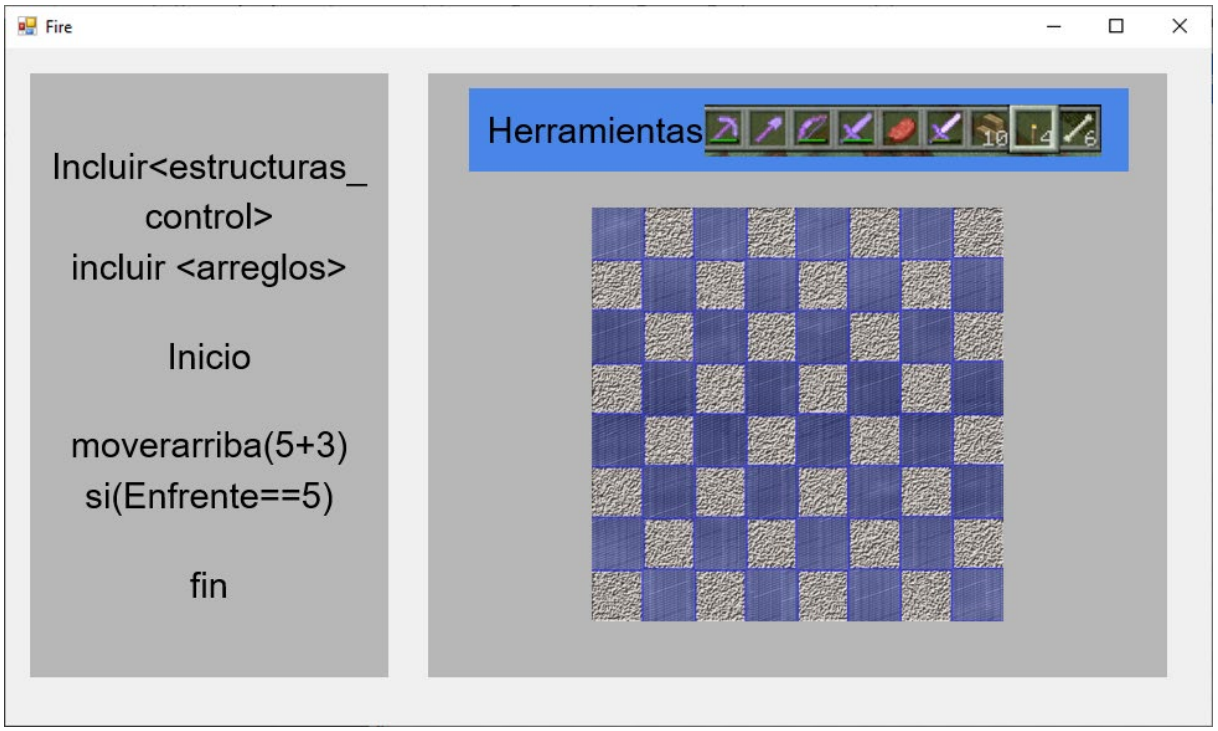

Figura 25 Boceto del espacio de trabajo

Lo anterior solo fue parte de lo que podría ser el proyecto final, es una idea que se irá desarrollando a lo largo de las siguientes páginas y posiblemente se desecharán algunas ideas anteriormente descritas (Spoiler: así sucederá). Pero esto es parte de la Ingeniería de Software.

4 Creación de un lenguaje de programación

En esta parte se diseñará un nuevo lenguaje de programación. Además, se definirán las bases que se deberán cubrir para el proyecto.

Para este capítulo trataré de explicar cómo es que se crea un lenguaje de programación basándose un poco en los lenguajes que se han visto en el *Capítulo 3.1 Estado del Arte* y en el lenguaje de programación C. Al final, se van a definir ciertas características de estos lenguajes estudiados.

El objetivo de diseñar un nuevo lenguaje de programación es crear un compilador que sea capaz de identificar este lenguaje y generar un código destino.

Existen distintas variables que se deben preguntar antes de crear un lenguaje de programación:

- ¿Se va a basar en uno existente?
- ¿Para qué se va a utilizar?
- ¿Se va a crear desde cero?
- ¿Partirá desde algún lenguaje ya establecido?
- ¿Cuál será el lenguaje objetivo?
- ¿Se va integrar a algo?
- ¿Qué paradigma de programación será?

Primero explicaré porqué se requiere crear un nuevo lenguaje de programación. En el Estado del Arte que se realizó encontré las siguientes observaciones

- Un lenguaje “arrastrar, soltar y configurar” no es para el grupo de edad al que quiero llegar. Hay que recordar que este proyecto está dirigido para que alumnos de media superior o superior aprendan a programar. Un lenguaje como este puede hacer que al momento de pasar a un lenguaje de alto nivel de programación dificulte su uso (Por lenguaje de alto nivel me refiero a Java, C, C++, C#, etc.).
- El código de programación es sencillo. Algunos de los lenguajes de programación de las herramientas anteriormente descritas son sencillas, estilo pseudocódigo.
- El código de programación es complicado. En algunos casos vistos anteriormente se tenía que programar inmediatamente en un lenguaje de programación de alto nivel.

Visto lo anterior, se propone lo siguiente para el lenguaje de programación que se creará:

- No será un lenguaje del estilo arrastra, suelta y configura.
- Será un lenguaje inspirado en C. Debido al público objetivo al que se intenta llegar, se puede subir el nivel de lo que se intenta crear. Será un híbrido entre pseudocódigo y algún lenguaje de alto nivel.
- Será fuertemente basado en instrucciones de Robomind y Karel.
- El paradigma de programación a cubrir será la Programación Imperativa.

4.1 Características de un lenguaje

Un lenguaje de programación de alto nivel usualmente tiene los siguientes elementos:

- Palabras reservadas. Que son palabras que no pueden ser utilizadas como identificadores en cualquier parte del código fuente.
- Identificadores. Son los nombres que reciben las variables. Estos identificadores podrán modificar su valor a lo largo del programa.
- Comentarios. Sirven para documentar el código o desechar parte del código sin tener que borrarlo.
- Operadores aritméticos. Se ocupan para operaciones aritméticas.
- Operadores lógicos. Son valores booleanos que comparan valores lógicos.
- Operadores de comparación. Se ocupan para comparar valores entre identificadores y/o valores.
- Operadores de asignación. Se ocupan para asignar valores a los identificadores.
- Delimitadores. Son usualmente elementos que ayudan a encapsular ciertas partes del código fuente, con el objetivo de tener un mayor orden.

Para este punto podría haber algunas cosas que no queden del todo claras, los ejemplos vienen un poco más adelante.

4.2 Características del lenguaje

Finalmente ha llegado el momento de comenzar a dar forma a el lenguaje.

Crear un lenguaje desde cero no tiene límites. Se puede hacer cualquier cosa que se posible, como que los identificadores sean solo números. Que los operadores lógicos se compongan de letras en lugar de símbolos. Olvidarse para siempre de los puntos y coma o llaves que abren y cierran. Pero antes de continuar, es necesario regresar unos párrafos atrás.

Como se dijo en las características que tendrá el lenguaje, el lenguaje a crear deberá ser basado en C, por lo que gran cantidad de elementos estarán inspirados en el lenguaje C. El siguiente recuadro será el primer ejemplo de código que se analizará en este trabajo.

El código de abajo es un programa en C, no importa lo que realiza sino sus elementos. Escribí todos los que se me ocurrieron. Cada uno de estos elementos deben ser tomados en cuenta para la creación del nuevo lenguaje.

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.
4.  #define pi 3.1416
5.
6.  void ejecutaMiFuncion(int);
7.
8.  void main(){
9.        int variableEntera = 80;
10.       float variableConDecimales, otraVariable;
11.
```

```
12.       //Aquí comienza la diversión
13.       variableConDecimales = 8.81;
14.
15.       /* Primer operación */
16.       otraVariable = variableConDecimales * (2 + variableEntera);
17.       //otraVariable = variableConDecimales * (otraVariable + variableEntera);
18.
19.       if(variableEntera==9){
20.                 printf("La variable es 9");
21.       }else if((variableEntera<9) && (variableEntera>=90) ){
22.                 ejecutaMiFuncion(4);
23.       }
24. }
25.
26. void ejecutaMiFuncion(int parametroDeEntrada){
27.       for(int i=0; i < parametroDeEntrada ; i++){
28.                 printf("Estoy en la linea 28 :P %d ",parametroDeEntrada);
29.       }
30. }
31.
```

Del extracto del código anterior se puede referenciar algunos elementos para el lenguaje.

Comenzando a analizar las primeras 5 líneas de código del ejemplo anterior.

- En estas primeras 4 líneas de código se puede encontrar directivas del preprocesador. Como refieren (Curiel Marquez & Larios Valencia, 2005) estas ayudan para indicar al compilador cómo preparar el contenido de un programa fuente antes de su traducción.
- En la Línea 6 se encuentra la declaración de una función. Este paso es opcional dependiendo del compilador que se utilice. En algunos casos si ésta declaración no se encuentra definida, al momento de compilar arrojará un error debido a que cuando se trata de utilizar la función se reconocerá como “no declarada”, aunque aparezca en la parte de abajo.

Ahora hay que detenerse a pensar lo siguiente: ¿lo anterior tiene sentido incluirse en el nuevo lenguaje?

Primero las directivas de preprocesador. Como se indicó, estas solo ayudan para indicarle al compilador que realice un previo análisis antes de realizar el proceso de compilación. Para el caso de este proyecto no será necesario incluirlo puesto que se está diseñando un compilador que genere instrucciones en tiempo de ejecución.

Por último, la declaración de funciones. Esta característica podría desecharse. Aunque es una característica bastante útil sobre todo para comprender la importancia de declarar absolutamente todo lo que se realiza, se va a desechar la idea de incluirlo en el nuevo lenguaje. Aunque eso significa que el lenguaje debe ser capaz de reconocer las funciones de otra manera.

Continuando con el análisis, el siguiente bloque de código se trata de la función principal “main” de C. Esta función es de vital importancia porque es la que indica la ejecución de

un programa cuando se ejecuta un programa. Esta función comienza en la línea 8 y termina en la línea 24.

- En la línea 8 se encuentran con distintos elementos. A continuación, intentaré detallar estos elementos:
    - El primer elemento que se encuentra es el tipo de función que es el valor que retornará el programa.
    - Seguido de la palabra reservada “main”, que es el nombre de la función principal. La función principal indica a los compiladores qué función será la primera que se ejecute, todos los lenguajes tienen una función principal.
    - Dentro de los paréntesis podrían llevar parámetros que recibe la función, para este caso no recibe de ningún parámetro.
    - Por último, se encuentra una llave que abre. Su objetivo es encapsular el bloque de código que puede llevar esa función. Este encapsulamiento termina en la línea 24.
- En las Líneas 9 y 10 se encontrará una construcción de un identificador. Acorde a (Menchaca García, 2002, p. 32) los identificadores se utilizan para designar los tipos de variables, campos, constantes simbólicas o funciones.
- En las Líneas 12, 15 y 17 se encontrarán comentarios. Estos son omitidos por el compilador pues se consideran espacios en blanco.
- En la Línea 13 se encuentra con una asignación, esta asignación se realiza cuando ya fue declarada la variable. Y se puede ejecutar más instrucciones como está a lo largo del programa.
- En la Línea 16 se encuentra una operación aritmética.
- En la Línea 19 se encuentra la primera estructura de control. Su bloque termina en la línea 23.
- Así mismo en la Línea 19 se encuentra una comparación de valores encerrada en paréntesis. La evaluación de estas siempre resultará en un booleano (falso o verdadero).
- En la Línea 20 se encuentra una función que permite mostrar una cadena en la consola.
- En la Línea 21 se encuentra que la estructura de control se trata de un “else if”.
- Así mismo en la Línea 21 se encuentra la primera comparación lógica, que compara valores booleanos, la respuesta a estas comparaciones igualmente resulta en un booleano.
- En la Línea 22 se tiene una llamada a función, con un parámetro que recibirá la función.

Una vez terminado el análisis, de nuevo hay que detenerse a analizar: ¿Qué elementos de estos pueden ayudar para la creación del lenguaje?

Comenzando con los elementos de la Línea 8:

- El tipo de funciones de una función sí importa y mucho. Pero para el proyecto que se planea realizar no. Ni siquiera se planea una instrucción como “return” para regresar el valor de la función. Y la razón es que no tiene sentido, más adelante explicaré a detalle el por qué no tiene sentido.
- La palabra “main” de una función indica que es la función principal, que es la primera en ejecutarse y que, a partir de ese punto, tomará forma el programa a ejecutar. Para el caso del lenguaje, sí, una palabra similar debe definirse. La razón es porque (spoiler) igualmente se planea agregar funciones y no se debe perder la función prioritaria entre las funciones.
- Los paréntesis son otro elemento que se incluirá en el lenguaje. Ayuda a encapsular muchas cosas, además de que se vuelve más ordenado el lenguaje.
- Las llaves que abren y cierran igualmente se incluirán en el lenguaje. Ayuda a delimitar las instrucciones de un bloque de código. Otros lenguajes de programación utilizan tabuladores, pero los tabuladores suelen ser confusos de manejar. Por lo que las llaves serán un elemento importante en el lenguaje a crear.

De igual manera se conservará la forma en la que se realizan las declaraciones como en las Líneas 9 y 10. Es decir, se deberá tener en cuenta la asignación de valores desde que se crea una variable o no. Hablando de la asignación de valores, igualmente se debe considerar que se puede signar valores a las variables creadas en cualquier momento del programa.

Los comentarios ayudan a documentar el código o encapsular código para no ejecutar, así que no veo el problema del por qué no deberían existir en el lenguaje.

Lo mismo para las operaciones aritméticas, no veo el por qué no deberían existir. Aquí quiero hacer un paréntesis: y es el tipo de dato que se va a manejar. Para este caso solo podrán convivir los números enteros. Se planea moverse a través de un tablero, no tiene sentido moverse 1.5 cuadros hacía arriba. Es la razón por la cual solo se trabajarán con números enteros. Si en estas operaciones el resultado llegase a ser un número flotante, se deberá redondear al entero más cercano.

Las estructuras de control son el objetivo de esta herramienta, no hay que olvidar que se trata de aprender a programar. En el ejemplo se observó un caso de “else if” pero se deben cubrir todas las estructuras posibles como las comparaciones “case” en C.

Las comparaciones lógicas y de valores tendrán que estar ahí. Agregarán mayor valor y complejidad al proyecto. Sobre todo, para entender cómo se utilizan.

La llamada a funciones será otra característica que le generará a este proyecto su valor. Así que se debe tomar en cuenta el soporte a funciones.

Por último, se analizará el último bloque de código que componen de la Línea 26 a la Línea 29:

- En la Línea 26 se encuentra los elementos de la función. Ya los se ha analizado con anterioridad salvo que, para este caso, sí incluye parámetros de entrada que deberán ser utilizados dentro de la función.
- En la Línea 27 se encuentra una estructura de control iterativa llamada “for”. Esta estructura es una de las que más he utilizado en mi carrera. Se observa que al final de la línea, tiene una llave que abre y que el bloque de código cierra en la Línea 29.
- En la Línea 28 se encuentra una función para imprimir en consola que es parecida a la Línea 20. Con la diferencia de que en esta función imprimirá un valor de una variable.

Una vez más se realiza el análisis: ¿qué elementos de estos sirven?

De la Línea 26 además de lo anteriormente planeado, se tiene que tomar en cuenta que las funciones tienen nombres distintos y que nombrarlas no debe ser una limitante. Además, el pase de parámetros debe estar ahí.

En la Línea 27 hay otro elemento clave, y es que el lenguaje debe soportar las estructuras de control iterativas como “for”, “while” y “do while”.

Y para finalizar, se debe considerar esos que algunos valores se imprimen en consola como en la Línea 28. Por lo que el lenguaje debe considerar la implementación de estos casos.

4.3 ¿Qué tomar en cuenta para el nuevo lenguaje?

Realizando un resumen del análisis anterior:

- Se debe considerar diferenciar la función principal de cualquier otra.
- Se deben considerar los paréntesis y llaves para hacer del lenguaje más ordenado.
- Se debe poder declarar variables, asignar valores durante su declaración o asignar valores durante la ejecución.
- La posibilidad de hacer comentarios en el código estará incluida.
- Se podrán realizar operaciones aritméticas básicas (Suma, Resta, Multiplicación y División).
- Debido a que para el proyecto solo tienen sentido los números enteros, se considera que aquellos números que debido a operaciones aritméticas resulten ser números reales, estos deberán ser sustituidos a su número real más cercano a este.
- Se contemplan las estructuras de control secuenciales “case” e “if else”.
- Se contemplan las estructuras de control iterativas como “do while”, “do” y “for”.
- Las comparaciones lógicas y las comparaciones de valores deberán estar soportadas para utilizarse en las estructuras de control.
- El soporte de funciones debe estar contemplado, así como el pase de parámetros a estas funciones.

- Las funciones pueden ser nombradas como al usuario final le sea más conveniente.
- Se tiene que contemplar la salida de texto. Ya sea para control, así como la posibilidad de que este texto incluya el valor de una variable como la salida de texto.

Parece que falta algo, ¿no? Sí. Durante la presentación del proyecto hablé algo sobre el soporte de arreglos. Sin embargo, para este punto, se excluye completamente el soporte para arreglos dimensionales y bidimensionales.

Aunque me hubiera encantado incluirlo, la idea no terminó por aterrizar. Quiero explicar por qué sucedió esto. Para comenzar se requiere regresar a la Figura 25: durante el concepto para este proyecto en la barra superior se observa ver una especie de herramientas. Esta idea debía ser algo como una especie de herramientas que puede utilizar el personaje (Aún por definir) y para acceder a estas herramientas se pretendía que el personaje pudiera acceder a ellas con comandos al estilo de arreglos dimensionales, pero esta idea la terminé desechando. Otra idea era hacer tal cual el soporte para arreglos dónde solo guardara números enteros, pero nuevamente esta idea no terminó de aterrizar para el proyecto así que nuevamente la deseché. El soporte a arreglos podría volverse a discutir para otra ocasión.

4.4 Nace un lenguaje

Retomando en el capítulo anterior, ahora se puede definir ahora un lenguaje de programación. A lo largo de este proyecto se utilizará la técnica “divide y vencerás” que consiste en separar un problema en general a problemas pequeños y de ahí partir hasta cubrir todos los requerimientos. (El capítulo anterior a este es en realidad la definición de requerimientos para el lenguaje).

4.4.1 El Mapa

Antes de continuar, es necesario definir cómo será el mapa de trabajo. Este mapa será el entorno en la cual se encuentran obstáculos que ayudarán a darle dinamismo a la herramienta. Con base a esto, se puede generar las respectivas instrucciones.

Hay que recordar que el tablero se verá más o menos de la siguiente manera:

#	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	X											B	B		T
1									Q				Q		T
2		B		Q			B			B			Q		T
3						Q							Q		T
4									O				Q		T
5					B							B	B		T
6		Q											Q		T
7							B						Q		T
8						Q							Q		T
9												B	B		T

Dónde:

O = Es la representación del personaje. Puede estar mirando en Norte, Sur, Este u Oeste. Q = Son obstáculos fijos con los que no puede interactuar el personaje. T = Son objetos que pueden ser tomados por el personaje. Además, se pueden eliminar. El personaje solo puede tomar un objeto a la vez. B = Son explosivos que, si se tocan, el usuario tendrá que volver a comenzar. Si se desactivan se vuelven objetos que pueden ser tomados y eliminados. X = Es la meta a dónde debe llegar el personaje.

Tomando en cuenta el modelo pasado, se tienen que escribir las respectivas instrucciones para manejar el personaje.

4.4.2 Función Principal

Comenzando con la parte más básica del programa, la definición de la función principal. Esta parte del código es la que siempre se tendrá que escribir. Para este concepto quedará de esta forma:

Español	Inglés
1. programa(){ 2. BLOQUE DE CÓDIGO 3. }	1. program(){ 2. CODE 3. }

Este nuevo lenguaje de programación estará en 2 idiomas: inglés y español.

En el recuadro anterior se tiene lo más básico y es prácticamente lo que el usuario final siempre deberá escribir, tal cual como se hace en todos los lenguajes de programación. La palabra “program” o “programa” será la palabra reservada que indica el nombre de la función principal.

4.4.3 Instrucciones del lenguaje

El siguiente paso es definir las instrucciones que podrá realizar el personaje. Recordando que el personaje se deberá mover sobre un tablero en las cuales podrá encontrar obstáculos. A continuación, se enlistan las posibles instrucciones que se utilizarán para moverse a lo largo del tablero (mapa).

Español	Inglés	Descripción
avanzar(n)	move(n)	Es la instrucción con la cual el personaje se moverá a lo largo del mapa en el entorno de trabajo.
espera(n)	wait(n)	Es la instrucción con la cual el personaje esperará una fracción de segundos para esquivar obstáculos.
gira_izquierda(n)	turn_left(n)	Es la instrucción con la cual el personaje girará 90° a la izquierda de acuerdo al sentido que esté mirando el objeto.
gira_derecha(n)	turn_right(n)	Es la instrucción con la cual el personaje girará 90° a la derecha de acuerdo al sentido que esté mirando el objeto.
toma_objeto()	pick_up()	Es la instrucción con la cual el personaje tomará algún objeto que se encuentre enfrente de él. El personaje lo cargará hasta ejecutar la función “dejar_objeto”.
soltar_objeto()	put_down()	Es la instrucción con la cual el personaje dejará algún objeto que haya cargado previamente con la instrucción “toma_objeto” dejándola en frente de dónde se encuentre.
eliminar_objeto()	delete_object()	Es la instrucción con la cual el personaje destruirá un objeto que se encuentre enfrente de él. Sólo podrá destruir objetos que se puedan destruir.
pintar_suelo()	paint_floor()	Esta instrucción servirá para realizar marcas en el “suelo” del tablero, una vez activado el objeto dejará marcas en el suelo hasta que se desactive con la instrucción “dejar_de_pintar”.
dejar_de_pintar()	stop_painting()	Esta instrucción servirá para dejar de realizar marcas en el suelo mientras el personaje avanza.
disable_bomb() disable_kaboom()	desactivar_bomba() desactivar_kaboom()	Esta instrucción deshabilitará una bomba y lo convertirá en un objeto manipulable. Si el personaje alcanza este objeto sin antes desactivarlo, se tendrá que iniciar de nuevo.
object_in_front()	tengo_objeto_delante()	Detecta si el personaje tiene un objeto que puede manipular en frente de él.

bomb_in_front() kaboom_in_front()	tengo_bomba_delante() tengo_kaboom_delante()	Detecta si el personaje tiene una bomba en frente de él.
wall_in_front()	tengo_muro_delante()	Detecta si delante del personaje tiene un muro, este objeto no es manipulable.
in_front_me()	delante_de_mi()	Detecta el objeto exacto que está delante el personaje. Puede ser objeto manipulable, un muro, o una bomba.
random()	aleatorio()	Regresa un número aleatorio. Se puede modificar la instrucción para obtener un número aleatorio entre un rango de números.

Estas instrucciones serán suficientes para continuar con el proyecto. Continuando con el programa que se comienza ahora se puede agregar lo siguiente:

Inglés	Español
1. program(){ 2. move(n); 3. pick_up(); 4. paint_floor(); 5. } 6.	1. programa(){ 2. avanzar(n); 3. toma_objeto(); 4. pintar_suelo(); 5. } 6.

Para este punto, el nuevo lenguaje de programación está tomando forma.

4.4.4 Variables y Símbolo de Asignación

haciendo un análisis de las variables, es importante recordar que las variables serán fundamentales para comprender los conceptos básicos de programación. Para declarar variables en algunos lenguajes es necesario declarar el tipo de dato que será (Entero o Flotante), para este caso no tiene sentido declararlas con “int” al inicio. Para este caso se ocupará la palabra “var” para decirle al compilador que esta por crear una variable. Este nombre es inspirado en “var” del lenguaje de programación PHP.

Antes de continuar, se analiza cómo se vería el lenguaje el símbolo de asignación. El símbolo de asignación será utilizado para darle valores a la variable creada. La forma de ingresar estos valores podrá ser al declararla, o durante la ejecución del programa. Además, los valores de asignación podrán ser dados de una operación aritmética o directamente de números enteros. El símbolo que se ocupará es original, es decir, creo que no se ha ocupado para algún otro lenguaje. El símbolo será una flecha apuntando a la variable, significa que se está asignando el valor a esa variable. El símbolo es este: “<-”. Es parecido a C cuando se estudia estructuras de datos, pero en este caso está al revés.

Tomando en cuenta estas 2 características anteriormente descritas, el concepto de lenguaje quedará de la siguiente manera:

Español	Inglés
1. programa(){ 2. var numero; 3. numero <- 8+96; 4. 5. avanzar(numero); 6. toma_objeto(); 7. pintar_suelo(); 8. } 9.	1. program(){ 2. var number; 3. number <- 8+96; 4. 5. move(number); 6. pick_up(); 7. paint_floor(); 8. } 9.

4.4.5 Estructuras de control iterativas y de control

Lo siguiente a tomar en cuenta son las estructuras de control, a continuación, se enlistan las distintas formas que podría tener una estructura de control:

Español	Inglés

1. si(«CONDICIÓN»){
2. «BLOQUE DE CÓDIGO»
3. }si_no{
4. «BLOQUE DE CÓDIGO»
5. }
6.
	1. if(«CONDICIÓN»){
	2. «BLOQUE DE CÓDIGO»
	3. }else{
	4. «BLOQUE DE CÓDIGO»
	5. }
	6.

1. comparar(«VARIABLE»){ 2. caso «CONDICIÓN»: 3. «BLOQUE DE CÓDIGO» 4. fin; 5. default: 6. «BLOQUE DE CÓDIGO» 7. fin; 8. } 9.	1. compare(«VARIABLE»){ 2. case «CONDICIÓN»: 3. «BLOQUE DE CÓDIGO» 4. end; 5. default: 6. «BLOQUE DE CÓDIGO» 7. end; 8. } 9.
1. repite_hasta(«CONDICIÓN»){ 2. «BLOQUE DE CÓDIGO» 3. } 4.	1. do_over(«CONDICIÓN»){ 2. «BLOQUE DE CÓDIGO» 3. } 4.
1. hacer{ 2. «BLOQUE DE CÓDIGO» 3. }mientras(«CONDICIÓN»); 4.	1. do{ 2. «BLOQUE DE CÓDIGO» 3. }while(«CONDICIÓN»); 4.
1. para(«INICIALIZACIÓN» ; «CONDICIÓN» ; «INCREMENTO»){ 2. «BLOQUE DE CÓDIGO» 3. } 4.	1. for(«INICIALIZACIÓN» ; «CONDICIÓN» ; «INCREMENTO»){ 2. «BLOQUE DE CÓDIGO» 3. } 4.

Tomando en cuenta lo anterior, observando cómo se vería en el concepto de lenguaje. Por simplicidad no se agregarán cada uno de los casos descritos anteriormente, sin embargo, sí se tomarán todos en cuenta.

Español	Inglés

1. programa(){
2. var numero;
3. numero <- 8+96;
4.
5. si(«CONDICIÓN»){
6. toma_objeto();
7. }
8.
9. hacer{
10. avanzar(numero);
11. pintar_suelo();
12. }mientras(«CONDICIÓN»);
13.
14. }
15.
	1. program(){
	2. var number;
	3. number <- 8+96;
	4.
	5. if(«CONDICIÓN»){
	6. pick_up();
	7. }
	8.
	9. do{
	10. move(number);
	11. paint_floor();
	12. }while(«CONDICIÓN»);
	13.
	14. }
	15.

4.4.6 Condiciones lógicas y de valores

El siguiente paso para el lenguaje es definir las condicionales que podrá llevar el programa. Hay que recordar que, para este caso, se trabajará con comparaciones lógicas y de valores.

Es decir, las lógicas las que comparan 2 booleanos, y las de valores las que comparan valores.

Para las comparaciones lógicas se ocupan los símbolos que representen las operaciones “and” y “or”. Para este proyecto “or” será representado por la concatenación de “or” y la operación “and” será representado por la concatenación de 3 letras “and”.

Para las comparaciones de valores numéricos se utilizarán las de toda la vida en C. Las cuales son: Son iguales (==), menor o igual que (<=), mayor o igual que (>=), menor que (<), mayor que (>), y diferente de (!=).

Para el concepto de lenguaje que se encuentra analizando, quedaría de la siguiente manera:

Español	Inglés

1. programa(){
2. var numero;
3. numero <- 8+96;
4.
5. si(numero < 89){
6. toma_objeto();
7. }
8.
9. hacer{
10. avanzar(numero);
11. pintar_suelo();
12. }mientras((numero < 89) or (78==78));
13.
14. }
15.
	1. program(){
	2. var number;
	3. number <- 8+96;
	4.
	5. if(number < 89){
	6. pick_up();
	7. }
	8.
	9. do{
	10. move(number);
	11. paint_floor();
	12. }while((number < 89) and (78==78));
	13.
	14. }
	15.

4.4.7 Funciones

Por último, se agregará las funciones que se plantearon anteriormente.

Las funciones no serán declaradas como sucede en C, pero tendrán que estar al final del código como una buena práctica de C.

A las funciones pueden utilizar el pase de parámetros de entrada para que sean utilizadas en la función.

Finalmente, el concepto de código se verá de la siguiente forma:

Español

```
1.  programa(){
2.        var numero;
3.        numero <- 8+96;
4.
5.        si(numero < 89){
6.                  toma_objeto();
7.        }
8.
9.        funcion(18);
10.
11. }
12.
13. funcion(var nuevoNumero){
14.       hacer{
15.                 avanzar(numero);
16.                 pintar_suelo();
17.       }mientras((numero < 89) //
    (nuevoNumero==78));
18. }
19.
```

Inglés

```
1.  program(){
2.        var number;
3.        number <- 8+96;
4.
5.        if(number < 89){
6.                  pick_up();
7.        }
8.
9.        function(18);
10.
11. }
12.
13. function(var newNumber){
14.       do{
15.                 move(number);
16.                 paint_floor();
17.       }while((number < 89) //
    (newNumber==78));
18. }
19.
```

4.5 Antes de continuar al Analizador Léxico

Con esto se ha finalizado la parte de la creación del lenguaje de programación. Este es un concepto y podría cambiar a lo largo del proyecto.

Es un paso necesario, y fue importante detenerse en esta parte pues será de mucha ayuda para pasar este concepto a él analizador léxico y el analizador sintáctico.

Se ha cumplido el objetivo de crear un lenguaje de programación inspirado en C, con un pequeño grado de dificultad pensado en el público objetivo.

El siguiente paso es retomar esta sintaxis para crear lo que será el Analizador Léxico. Una vez terminado el Analizador Léxico se puede continuar con este mismo boceto para generar el Analizador Sintáctico.

5 La interfaz del proyecto

Antes de continuar al Analizador Léxico me gustaría crear la interfaz de usuario del programa. Esta interfaz será la principal del proyecto en sí.

La idea de trabajar sobre la interfaz misma del proyecto es que conforme se avance en el proyecto no se tenga que regresar para integrarlo al proyecto final.

Los pasos que se siguieron para crear un prototipo funcional se encuentran en el **Capítulo 1** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado.**

Al final el programa se visualizará de la siguiente manera:

Figura 26 Interfaz de usuario del Entorno de Desarrollo

Este será el prototipo (el cascarón) del programa, a partir de este prototipo se trabajará con los siguientes pasos.

Si se desea conocer el código fuente del prototipo dirigirse a la siguiente dirección web:

Enlace 1 Código fuente del prototipo https://github.com/JonathanGarcia15/tesis-01-primer-prototipo-de-un-compilador	

6 Analizador Léxico

En esta parte se generará el Analizador Léxico. Este analizador servirá de base para crear el verdadero Analizador Léxico que utilizará el compilador final. La razón de hacerlo así es para mostrar en este proyecto cómo se genera un Analizador Léxico con la herramienta JFlex.

Al final de este capítulo, se puede ver de forma escrita cada token que genera un Analizador Léxico y sentaremos algunas bases de cómo se trabajará en este proyecto.

Si se está ocupando esta tesis como una pequeña guía de apoyo, se tiene que saber que este pequeño analizador que se está a punto de generar será modificado casi en su totalidad en el próximo capítulo.

6.1 Creación de Expresiones Regulares para un Lenguaje de Programación

El primer paso para crear un Analizador Léxico es generar las expresiones regulares del lenguaje planteado. Gracias a estas expresiones regulares el Analizador Léxico reconocerá esos patrones de caracteres del programa fuente.

Para crear las expresiones regulares hay que ser bastante minucioso de no dejar ningún cabo suelto. Es decir, que se debe cubrir hasta el último detalle del lenguaje a crear.

Las expresiones regulares para un lenguaje son de las más sencillas de crear, pues los únicos patrones a identificar son palabras ya establecidas, a excepción de algunos casos que se analizarán más adelante.

Sin más, estas son las expresiones regulares que se ocupará para el lenguaje. Debido a que no existe un límite y para ser más ordenado se crearán expresiones regulares de cada elemento existente del lenguaje, por ejemplo, cada llave que abre tendrá una expresión regular individual y será distinta a la expresión regular que identifica una llave que cierra. Separar las expresiones regulares en muy pequeños fragmentos más detallados, facilitará algunos procesos en el próximo paso: El Analizador Sintáctico.

La notación de las expresiones ya se revisó en *Capítulo 1.5.5.3: Notación de las expresiones regulares*, aquí se agrega un símbolo más: la comilla doble (") esta ayudará a encapsular 1 o más símbolos. Son de utilidad para separar puntualmente los símbolos, y debido a que no existe restricción en utilizarlas podrá parecer que se abusa en el uso de este símbolo, pero ayudará enormemente a evitar errores al momento de pasar la expresión regular a la herramienta del Analizador Léxico.

Por último, quiero recordar que el programa estará soportado en inglés y español, por lo tanto, las expresiones regulares se están tomando en cuenta para este soporte de idiomas.

Comenzando con las expresiones regulares de las palabras reservadas que componen el programa más básico visto en el capítulo anterior.

Nombre de la expresión regular (Token)	Expresión Regular
PROGRAMA	program|programa
PARENTESISABRE	"("
PARENTESISCIERRA	")"
LLAVEABRE	"{"
LLAVECIERRA	"}"

El siguiente paso es definir las expresiones regulares de lo que serán las instrucciones del programa que podrá utilizar el personaje

Nombre de la expresión regular (Token)	Expresión Regular
AVANZAR	move|avanzar
ESPERA	wait|espera
IZQUIERDA	turn_left|gira_izquierda
DERECHA	turn_right|gira_derecha
TOMAR	pick_up|toma_objeto
SOLTAR	put_down|soltar_objeto
ELIMINAR	delete_object|eliminar_objeto
KABOOMDEFRENTE	bomb_in_front|tengo_bomba_delante |kaboom_in_front|tengo_kaboom_delante
PINTAR	paint_floor|pintar_suelo
DEJAPINTAR	stop_painting|dejar_de_pintar
OBJETODELANTE	object_in_front|tengo_objeto_delante
MURODELANTE	wall_in_front|tengo_muro_delante
QUETENGODELANTE	in_front_me|delante_de_mi
DESACTIVARKABOOM	disable_bomb|desactivar_bomba |disable_kaboom|desactivar_kaboom
RANDOM	random|aleatorio

Continuando con los operadores aritméticos, de comparación y lógicos que utilizará el lenguaje

Nombre de la expresión regular (Token)	Expresión Regular
OPERADORDECOMPARACION	"=="|"!="|"<="|">="|"<"|">"
OPERADORLOGICO	"or"|"and"
OPERADORNEGACION	"!"
OPERADORARITMETICO	"*"|"\/"
OPERADORARITMETICORESTA	"\-"
OPERADORARITMETICOSUM	"\+"
BOOLEANO	true|false|verdadero|falso

Ahora se añaden las expresiones regulares de las palabras que definen a las estructuras de control

Nombre de la expresión regular (Token)	Expresión Regular
SI	if|si
SINO	else|si_no
COMPARAR	compare|comparar
CASO	case|caso
FIN	end|fin
DOSPUNTOS	":"
PARA	for|para
DEFAULT	default
SINOSI	else_if|sinosi
REPITEHASTA	do_over|repite_hasta
REPITE	while|mientras
HACER	do|hacer

No olvidar las expresiones para los operadores de asignación, números, y otros elementos que podrían estar en el lenguaje.

Nombre de la expresión regular (Token)	Expresión Regular
OPERADORASIGNACION	"<-"
SEPARADOR	","
PUNTOYCOMA	";"
VARIABLE	variable|var
IMPRIMIRVARIABLE	print_var|imprimir_var|print_variable|imprimir_variable
IMPRIMIRCADENA	print|imprimir

Hay unas instrucciones de las cuales no he hablado, y es que durante el desarrollo del proyecto se agregaron algunas instrucciones más.

La primera de ellas son las funciones de terminar programa y terminar función. En el momento que se ejecuta el programa o función detienen su ejecución, es el equivalente a un “return” en main.

La segunda de ellas es una paleta de colores, la función “pintar suelo” necesitaba de este token para no confundirlo con un identificador.

Tomando en cuenta lo anterior, se crearon las siguientes expresiones regulares:

Nombre de la expresión regular (Token)	Expresión Regular
COLORES	azul|blue|green|verde|amarillo|yellow|red|rojo
TERMINARBLOQUE	finish|terminar

Todas las expresiones regulares que se han planteado no han sido “difíciles”, pues solo detectan un patrón específico.

Las siguientes expresiones regulares son las únicas que requirieron de algo más a lo que se ha estado creando. Aun así, son sencillas de definir.

Nombre de la expresión regular (Token)	Expresión Regular
COMENTARIO	"..." [^...]* "..." | "//" [^"\n"]* "\n"
NUMERO	[0-9]+
IDENTIFICADOR	[a-zA-Z|"_"][a-zA-Z|0-9|"_"]*
CADENA	"\"" [^"\""]* "\""

Para finalizar, es necesario incluir algunos caracteres que son sencillos pero que no se ven. Cada espacio en blanco, saltos de línea o tabuladores deben considerarse. De no hacerlo, el Analizador Léxico arrojará errores cada que se encuentre con estos símbolos. Esto puede hacer que a simple vista no se detecte el error, pero es necesario incluirlos.

Nombre de la expresión regular (Token)	Expresión Regular
SALTOLINEA	"\n"
BLANCO	[|\t|\r]+

6.2 Modificaciones al Entorno de Desarrollo

Antes de continuar, se requieren colocar algunos archivos dentro del Entorno de Desarrollo, para ver las modificaciones dirigirse al **Capítulo 2 y Sección 2.1** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado.**

6.3 La estructura de JFlex

Para crear el archivo de JFlex acorde a la documentación de JFlex (Klein, Rowe, & Décamps, 1998) se debe tomar en cuenta la siguiente estructura:

```
1.  «Código del usuario»
2.  %%
3.  «Opciones y declaraciones»
4.  %%
5.  «Reglas del Léxico»
6.
```

Antes de continuar, se requiere modificar una vez más el Entorno de Desarrollo, para ver las modificaciones dirigirse al **Capítulo 3** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado.**

6.4 La estructura del Analizador Léxico para JFlex

Ahora sí, se comenzará a llenar lo que será el Analizador Léxico. Este será encargado de leer un programa de entrada y comprobar que todos los elementos de ese texto puedan ser reconocidos por el Analizador Léxico.

Para esto se debe tener a la mano las expresiones regulares que ya se han creado y se han descrito en el *Capítulo 6.1: Creación de Expresiones Regulares para un Lenguaje de Programación*

Regresando al *Capítulo 6.3: La estructura de JFlex* dónde se define la estructura básica del archivo para este proyecto. El primer cambio es indicar el *package* a dónde pertenece e importar la clase *Tokens* con todos sus métodos, estos serán colocados dónde se indica que irá el código del usuario, quedará de la siguiente manera:

```
1.  package Compilador.Default.AnalizadorLexico;
2.  import static Compilador.Default.AnalizadorLexico.Tokens.*;
```

Después se toma una muestra de las expresiones regulares, simplemente se toma el nombre y la expresión regular y se agrega un operador de asignación. Todas las expresiones regulares que se crearon deberán tener el siguiente formato:

```
1.  COLORES=azul|blue|green|verde|amarillo|yellow|red|rojo
2.  IDENTIFICADOR=[a-zA-Z|"_"][a-zA-Z|0-9|"_"]*
3.  OPERADORDECOMPARACION="=="|"!="|"<="|">="|"<"|">"
```

No importa el orden en el que estén, lo único que importa es que estén todas las expresiones regulares que *se crearon en el Capítulo 6.1*

Después de colocar las expresiones regulares, se tiene que definir qué hacer con ellas cada que el Analizador Léxico las lea, para esto se utilizará la siguiente estructura:

```
1.  {«NOMBRE DE LA EXPRESIÓN REGULAR»} {
2.      «CÓDIGO DEL USUARIO»
3.      return «NOKMBRE DEL TOKEN»;
4.  }
```

Estas estructuras deberán de ir justo debajo de las expresiones regulares. Para el ejemplo de las expresiones regulares con nombres “COLORES”, “IDENTIFICADOR” y “OPERADOR DE COMPARACION” quedarán de la siguiente manera:

```
1.  {COLORES} {
2.      return COLORES;
3.  }
4.  {IDENTIFICADOR} {
5.      return IDENTIFICADOR;
6.  }
7.  {OPERADORDECOMPARACION} {
8.      return OCOMPARACION;
9.  }
```

Para este caso sí importa el orden en el que coloquen estos bloques de código, si al momento de compilar el Analizador Léxico arroja la siguiente advertencia:

```
Rule can never be matched:
```

Significa que se tiene que regresar a este punto para volver a acomodar estos bloques. Ya que JFlex acomoda estos bloques en una estructura de control llamada “case” y hay

que recordar que: si se encuentra con un “match” antes de llegar al “match” que se desea llegar, cada “match” contiene al final la palabra “break” que para lo que sirve esta instrucción es ya no continuar evaluando las siguientes condiciones del switch case. Es decir, el compilador de Java ya no seguirá comprobando qué hay “más abajo” ya que se saldrá de la estructura “Case” por lo que nunca podría llegar a ejecutarse ese bloque de código.

Por último, hay 1 expresión que se debe agregar a estos bloques, este bloque debe ir al final por la razón que platiqué en el párrafo anterior, y se trata de un bloque de error. Cuando el Analizador Léxico se encuentra con un símbolo que no se encuentra definido en sus expresiones regulares, se tendrá que tener una forma de poder controlar este error. Es decir, con esto se controla el error de un símbolo no reconocido, de no hacerlo el programa podría fallar cerrándose cuando suceda una excepción. Este bloque es el siguiente:

```
1.  . {
2.      return ERROR;
3.  }
```

Igualmente, para este ejercicio algunos tokens se pueden ignorar como el caso de los blancos que incluyen los espacios en blanco entre palabras y tabuladores.

```
1.  {BLANCO} {/* Blanco */}
```

Se puede realizar lo mismo para el caso de los caracteres Blancos para los saltos de línea, pero en este Analizador Léxico se dejará para comprobar correctamente los tokens. Esto es cuestión de diseño y dependerá de lo que se esté creando, debido a que los saltos de línea en algunas ocasiones tienen una función importante.

Al final, el archivo Lexer.flex de debería ver de la siguiente manera:

```
1.  package Compilador.Default.CodigoFuente;
2.  import static Compilador.Default.CodigoFuente.Tokens.*;
3.
4.  %%
5.
6.  %class Lexer
7.  %type Tokens
8.
9.  OPERADORASIGNACION="<-"
10. OPERADORLOGICO="//"|"&&"
11.
12. {SALTOLINEA} {
13.     return SALTOLINEA;
14. }
15.
16. . {
17.     return ERROR;
18. }
```

Por cuestiones prácticas, no colocaré todo el código debido a que son más de 250 líneas de código, si se desea consultar el resultado final de este archivo se deberá consultar la siguiente dirección:

Enlace 2 Resultado final del archivo Lexer.flex https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/src/Compilador/Default/CodigoFuente/Lexer.flex	

6.5 El Archivo *Tokens.Java*

El siguiente paso a la creación del compilador, será simplemente definir los tokens que se escribieron en el Analizador Léxico. Para esto simplemente se agrega la palabra "enum" de la clase Tokens quedando de la siguiente manera:

```
1.  package Compilador.Default.CodigoFuente;
2.  public enum Tokens {
3.  }
```

Ahora regresando a los bloques de código anteriores, y se copian todos los valores de retorno y se enlistan en la clase.

```
10. {COLORES} {
11.     return COLORES;
12.       }
```

Al final quedará la clase de la siguiente manera:

```
1.  package Compilador.Default.CodigoFuente;
2.  public enum Tokens {
3.  COLORES,
4.  SALTOLINEA,
5.  SOLTAR
6.  }
```

Igualmente, por cuestiones de practicidad, no colocaré la clase completa, pero se puede ver el resultado final consultando la siguiente dirección:

Enlace 3 Resultado final del archivo Tokens.java https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/src/Compilador/Default/AnalizadorLexico/Tokens.java	

6.6 Modificaciones al proyecto para generar un Analizador Léxico

Antes de realizar pruebas al Analizador Léxico construido es necesario realizar algunos pasos previos, estos pasos se encuentran en el **Capítulo 4** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado.**

6.7 Analizador Léxico en pruebas

En el **Capítulo 4** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado** se ha estado construyendo lo necesario para llegar a este punto, el siguiente paso es la prueba del Analizador Léxico.

Comenzando a probar cada uno de los elementos que se ha propuesto, el orden aquí no importa. Lo importante es que aparezcan aquí todos los elementos que se ha propuesto.

Por ejemplo, para el siguiente código fuente de ejemplo:

```
1.  programa(){
2.        <-
3.        or and
4.        == != <= >= < >
5.        !
6.        * /
7.        -
8.        +
9.        "Soy Una Cadena "
10.       ,
11.       ;
12.       ... Soy Un Comentario
13.
14.                turn_right gira_derecha ...
15.
16.       /// Soy Otro Comentario
17.
18.
19.       45 645645645
20.
21.       _SoyUnIdentificador
22.       _____
23.
24.       {
25.       }
26.       (
27.       program programa
28.       variable var
29.       )
30.       move avanzar
31.       wait espera
32.       turn_left gira_izquierda
33.       turn_right gira_derecha
34.       pick_up toma_objeto
35.       put_down soltar_objeto
36.       delete_object eliminar_objeto
37.       paint_floor pintar_suelo
38.       stop_painting dejar_de_pintar
39.       if si
40.       else si_no
41.       compare comparar
42.       case caso
43.       end fin
44.       :
```

```
45.       for para
46.       default
47.       else_if sinosi
48.       do_over repite_hasta
49.       while mientras
50.       do hacer
51.       azul blue green verde amarillo yellow red rojo
52.       finish terminar
53.       print_var imprimir_var print_variable imprimir_variable
54.       print imprimir
55.       object_in_front tengo_objeto_delante
56.       bomb_in_front tengo_bomba_delante kaboom_in_front tengo_kaboom_delante
57.       wall_in_front tengo_muro_delante
58.       in_front_me delante_de_mi
59.       disable_bomb desactivar_bomba disable_kaboom desactivar_kaboom
60.       true false verdadero falso
61. }
62.
```

Si se requiere conocer el código del programa de prueba se puede acceder a la siguiente dirección web:

Enlace 4 Consultar el código con el que se prueba el Analizador Léxico https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/OtrosArchivos/ProgramaDePrueba.txt	

Al ejecutarlo, debería ser capaz de reconocer cada una de la secuencia de caracteres. Para esto, solo hay que seguir la ruta de submenús “{*CÓDIGO*}” y “{*COMPILAR*}”. Y el resultado será el siguiente:

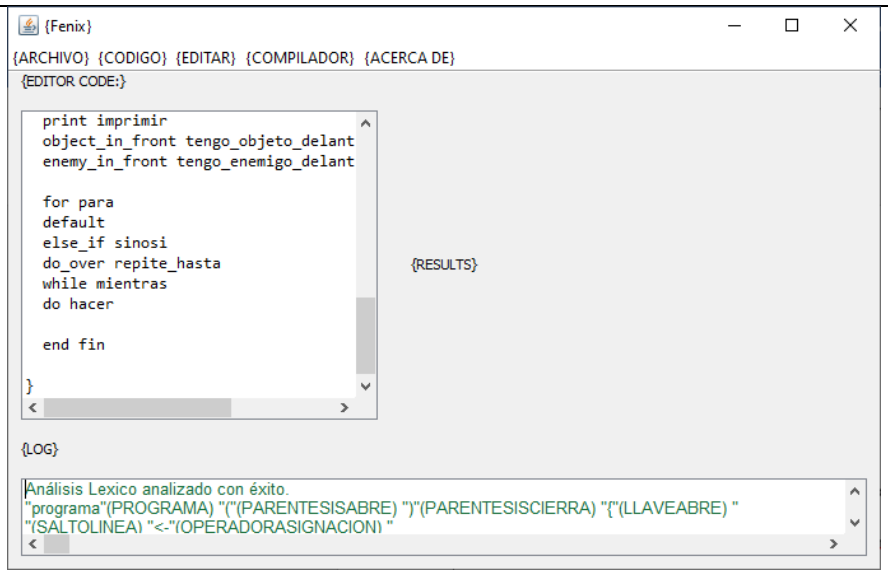

Figura 27 Resultado exitoso del Analizador Léxico

Si se requiere conocer completamente la cadena que aparece en color verde en la imagen anterior, se puede consultar el siguiente enlace:

Enlace 5 Consultar el código resultante después de probar el Analizador Léxico https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/OtrosArchivos/AnalisisExitoso.txt	

Por otro lado, por un solo carácter que no pueda reconocer el Analizador Léxico provocará un error. Esto se puede probar colocando un símbolo “?” o “=” en cualquier lado que no sea dentro de un comentario.

Indicará el símbolo que no se ha reconocido y el número de línea dónde se encuentra.

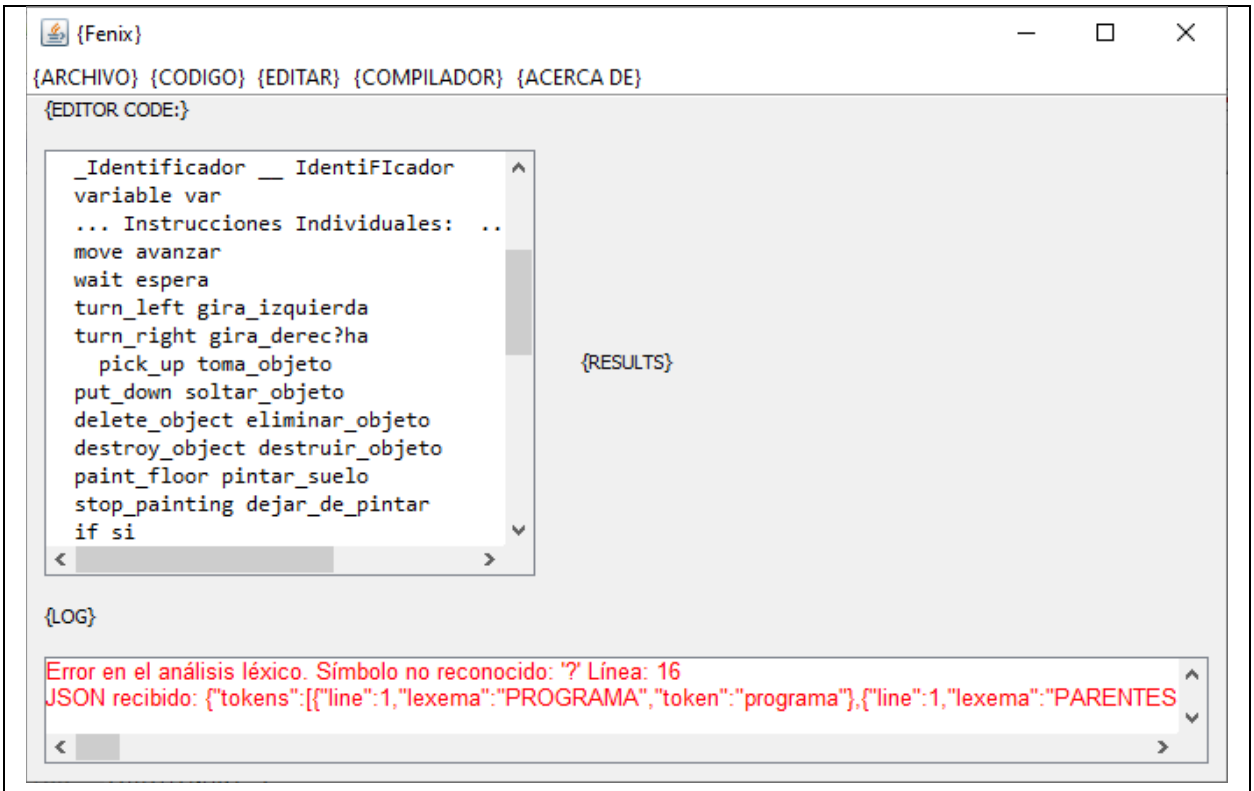

Figura 28 Resultado de error del Analizador Léxico

Si se requiere conocer completamente la cadena que aparece en color verde, consultar el siguiente enlace:

Enlace 6 Código completo del programa de prueba que garantiza un error léxico https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/OtrosArchivos/ProgramaDePruebaNoExitoso.txt	
Enlace 7 Código completo del resultado https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/OtrosArchivos/ErrorDeAnalisis.txt	

6.8 Archivos del proyecto

Así finaliza este capítulo sobre el Analizador Léxico, nuevamente si se desea descargar el proyecto hasta este punto se puede ingresar a la siguiente dirección web. Dentro de este proyecto irá incluida una carpeta “Otros Archivos”, esta no pertenece al proyecto, sino que dentro están los 2 archivos con las cadenas de mensajes de error o éxito descritas en el subtema anterior. Si alguien desea descargar este proyecto, esa carpeta se puede eliminar con toda seguridad.

Enlace 8 Código completo del Analizador Léxico https://github.com/JonathanGarcia15/tesis-02-analizador-lexico	

7 Analizador Sintáctico

Se ha llegado a la parte de generar el Analizador Sintáctico. Para este capítulo es importante que se haya entendido cómo generar un Analizador Léxico. No se generará uno desde el inicio, pero se ocupará lo anteriormente creado para modificarlo y generar otro distinto.

La razón de este cambio es que se puedan comunicar ambos analizadores el uno al otro, sin desechar lo que ya se ha realizado.

Al final de este capítulo, ya casi estará terminado el compilador. Aunque faltarán algunos detalles que explicaré más adelante. Pero al finalizar este capítulo se habrá creado un compilador completamente funcional, aunque con ausencia de algunas características.

7.1 Gramáticas Libres de Contexto

Lo primero que se debe tener antes de pasar a las herramientas que generarán el Analizador Sintáctico es tener las Gramáticas Libres de Contexto.

En el **Capítulo 6** de este documento, cuando se realizó la prueba del Analizador Sintáctico este no valoraba si, por así decirlo, tenía sentido lo que estaba leyendo. Es decir, solo se dedicó a dividir el texto en tokens e indicar que el programa fuente era o no, válido. Bueno, ahora es turno de crear esas reglas gramaticales para conocer si se está escribiendo de forma correcta un lenguaje de programación.

El primer paso de este proceso es imaginarse: ¿cuál será la forma final del lenguaje de programación con mucho más detalle? En este punto se debe detallar con exactitud cada variación que podría tener el lenguaje.

Retomando el programa básico, en el *Capítulo 4.4.2: Función Principal* dije que el programa iba a tener una función principal. Es decir, algo que el programador siempre tendrá que escribir. Sucede en lenguajes como Java, C, C#, C++. Siempre hay código tan repetitivo que tarde o temprano algún IDE lo termina escribiendo de forma automática para ahorrar un poco de trabajo.

En el caso del lenguaje, este será el código que siempre se debe escribir:

```
1.  programa(){
2.       « CÓDIGO DEL USUARIO »
3.  }
```

Partiendo de esto se construirá la gramática para ese nuevo lenguaje.

Hay que recordar que un Analizador Sintáctico generará un árbol semántico. Si se compara con un árbol de la vida real: las ramas serán los símbolos no terminales, y las hojas serán los símbolos terminales.

Primero hay que definir un Símbolo Inicial, usualmente se ocupa solo la letra “S” de “Start”. En este caso seré más expresivo esperando ser más claro con la gramática. El símbolo inicial será “Inicio”.

Después del arduo trabajo de definir el símbolo inicial, es momento de abstraer cuáles son los símbolos terminales del lenguaje. Los símbolos terminales, como su nombre lo indica, son aquellos patrones dónde termina el análisis (La hoja de los árboles). En el caso del programa ejemplo anterior hay varios:

- “programa” es un terminal que se ha definido como “PROGRAMA” en el léxico.
- Los paréntesis que abren y cierran son otros terminales.
- Las llaves que abren y cierran son terminales.

La gramática está comenzando a tener forma, se observa cómo se visualiza en texto:

```
Inicio ::= PROGRAMA PARENTESISABRE PARENTESISCIERRA LLAVEABRE LLAVECIERRA
```

Convenientemente se estará usando algunos símbolos propios de la herramienta CUP como “::=”. Además, aquí es cuando tiene sentido haber separado símbolo por símbolo en el Analizador Léxico, debido a que ya se conoce cuál símbolo ha leído. Por ejemplo, si se hubiera puesto únicamente “PARÉNTESIS” con una expresión regular “ (|) ”, en este paso se tendría que estar pensando en cómo reconocer que el paréntesis se está escribiendo de forma correcta. Es decir, validar que hay un paréntesis que abre y otro que cierra. Al haber separado cada símbolo se ha facilitado por mucho, esa tarea de reconocimiento.

Ahora el siguiente paso es pensar en cómo integrar el resto de instrucciones a la gramática. Se podría colocar en la gramática de “Inicio”, aunque si lo se piensa, no tiene mucho sentido, porque al final, se tendría un montón de instrucciones y eso provocaría un completo caos. Para esto se tiene a los Símbolos No Terminales. Los símbolos no terminales ayudarán a seguir creando más gramáticas muchísimo más completas y complejas.

Continuando con el ejemplo de la primera gramática, ahora se requiere solamente de indicar dónde estará el símbolo no terminal. Para diferenciar los símbolos terminales de los no terminales, a los símbolos no terminales se escribirán en minúsculas, mientras que los símbolos terminales serán escritos en mayúsculas. Este nuevo no terminal irá entre las llaves que abren y cierran, tendrá el nombre de “BloqueDeInstrucciones”

```
Inicio ::= PROGRAMA PARENTESISABRE PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones LLAVECIERRA
```

Pareciera que hasta aquí se ha terminado con la primera gramática, pero no. Aún falta un pequeño análisis más. Hay que recordar que se ha propuesto que el lenguaje soportará funciones. Si se realiza un análisis de los lenguajes de programación, las funciones usualmente están fuera del programa principal, esperando a ser llamadas. No detallaré exactamente cómo es que se tendría que ver una función (Lo haré más

adelante), pero es importante colocarla en este momento para conocer la gramática final del símbolo inicial.

Tomando como referencia el lenguaje C, los bloques de funciones se encontrarán al final de declarar la función principal. No se declararán las funciones. Por lo que el No Terminal deberá ser colocado al final de la primera gramática. Este no terminal se llamará "Funciones".

```
1.  Inicio ::= PROGRAMA PARENTESISABRE PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones  LLAVECIERRA
    Funciones
2.  ;
```

Con esto se ha terminado la producción de la primera regla gramatical.

La continuación y detalle de cada una de las producciones de la gramática para el lenguaje de programación se encuentran en el **Apéndice 2: Detalle de la Gramática Libre de Contexto.**

Sin embargo, la gramática completa del lenguaje es el siguiente:

```
 1. start with Inicio;
 2.
 3. Inicio::=
 4.      PROGRAMA PARENTESISABRE PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones  LLAVECIERRA
Funciones
 5. ;
 6.
 7. BloqueDeInstrucciones::=
 8.      InstruccionDeclaracionDeVariables BloqueDeInstrucciones
 9.      | InstruccionModificacionDeValorDeVariables BloqueDeInstrucciones
10.      | InstruccionAvanzar BloqueDeInstrucciones
11.      | InstruccionEspera BloqueDeInstrucciones
12.      | InstruccionGirarALaIzquierda BloqueDeInstrucciones
13.      | InstruccionGirarALaDerecha BloqueDeInstrucciones
14.      | InstruccionTomarObjeto BloqueDeInstrucciones
15.      | InstruccionSoltarObjeto BloqueDeInstrucciones
16.      | InstruccionEliminarObjeto BloqueDeInstrucciones
17.      | InstruccionDesactivarKaboom BloqueDeInstrucciones
18.      | InstruccionPintarSuelo BloqueDeInstrucciones
19.      | InstruccionDejarDePintarSuelo BloqueDeInstrucciones
20.      | InstruccionImprimirVariables BloqueDeInstrucciones
21.      | InstruccionImprimirCadenas BloqueDeInstrucciones
22.      | InstruccionTengoObjetoDelante BloqueDeInstrucciones
23.      | InstruccionTengoBombaDelante BloqueDeInstrucciones
24.      | InstruccionQueTengoDelanteDeMi BloqueDeInstrucciones
25.      | TengoMuroDelanteDeMi BloqueDeInstrucciones
26.      | InstruccionTerminarBloque BloqueDeInstrucciones
27.      | EstrucutraDeControlIf BloqueDeInstrucciones
28.      | EstrucutraDeControlCase BloqueDeInstrucciones
29.      | EstrucutraDeControlFor BloqueDeInstrucciones
30.      | EstrucutraDeControlDoWhile BloqueDeInstrucciones
31.      | EstrucutraDeControlWhile BloqueDeInstrucciones
32.      | LlamadaAFuncion BloqueDeInstrucciones
33.      | /* Epsilon: Sin Bloque de Instrucciones */
34. ;
35.
36. InstruccionDeclaracionDeVariables::=
37.      VARIABLE DeclaracionDeVariables PUNTOYCOMA
```

```
38. ;
39.
40. DeclaracionDeVariables::=
41.      IDENTIFICADOR
42.      | IDENTIFICADOR AsignacionDeValoresAlDeclararVariable
43.      | IDENTIFICADOR MultiplesDeclaraciones
44. ;
45.
46. MultiplesDeclaraciones::=
47.      SEPARADOR IDENTIFICADOR MultiplesDeclaraciones
48.      | SEPARADOR IDENTIFICADOR AsignacionDeValoresAlDeclararVariable MultiplesDeclaraciones
49.      | SEPARADOR IDENTIFICADOR AsignacionDeValoresAlDeclararVariable
50.      | SEPARADOR IDENTIFICADOR
51. ;
52.
53. AsignacionDeValoresAlDeclararVariable::=
54.      OPERADORASIGNACION NUMERO
55.      | OPERADORASIGNACION IDENTIFICADOR
56.      | OPERADORASIGNACION NumeroAleatorio
57.      | OPERADORASIGNACION OperacionAritmetica
58. ;
59.
60. OperacionAritmetica::=
61.      OperacionSinParentesis
62.      | OperacionConParentesis
63. ;
64.
65. OperacionConParentesis::=
66.     PARENTESISABRE OperacionAritmetica PARENTESISCIERRA continuaOperacion
67. ;
68.
69. continuaOperacion::=
70.     ContinuacionOperacionSinParentesis
71.     | /* No continua nada después de un paréntesis que cierra */
72. ;
73.
74. OperacionSinParentesis::=
75.      IDENTIFICADOR ContinuacionOperacionSinParentesis
76.      | NUMERO ContinuacionOperacionSinParentesis
77.      | NumeroAleatorio ContinuacionOperacionSinParentesis
78. ;
79.
80. ContinuacionOperacionSinParentesis::=
81.      OperadoresAritmeticos IDENTIFICADOR ContinuacionOperacionSinParentesis
82.      | OperadoresAritmeticos NUMERO ContinuacionOperacionSinParentesis
83.      | OperadoresAritmeticos NumeroAleatorio ContinuacionOperacionSinParentesis
84.      | OperadoresAritmeticos OperacionConParentesis
85.      | OperadoresAritmeticos IDENTIFICADOR
86.      | OperadoresAritmeticos NUMERO
87.      | OperadoresAritmeticos NumeroAleatorio
88. ;
89.
90. OperadoresAritmeticos::=
91.      OPERADORARITMETICO
92.      | OPERADORARITMETICOSUMA
93.      | OPERADORARITMETICORESTA
94. ;
95.
96. InstruccionModificacionDeValorDeVariables::=
97.      IDENTIFICADOR AsignacionDeValoresAlDeclararVariable PUNTOYCOMA
98. ;
99.
100. InstruccionAvanzar::=
101.      AVANZAR PARENTESISABRE ParametrosDeEntradaDeUnaInstruccion PARENTESISCIERRA PUNTOYCOMA
102. ;
```

```
103.
104. NumeroAleatorio::=
105.     RANDOM PARENTESISABRE NUMERO SEPARADOR NUMERO PARENTESISCIERRA
106.     | RANDOM PARENTESISABRE PARENTESISCIERRA
107. ;
108.
109. ParametrosDeEntradaDeUnaInstruccion::=
110.     IDENTIFICADOR
111.     | NUMERO
112.     | NumeroAleatorio
113.     | OperacionAritmetica
114.     | /* Los parámetros de entrada pueden ir vacíos */
115. ;
116.
117. InstruccionEspera::=
118.     ESPERA PARENTESISABRE ParametrosDeEntradaDeUnaInstruccion PARENTESISCIERRA PUNTOYCOMA
119. ;
120.
121. InstruccionGirarALaIzquierda::=
122.     IZQUIERDA PARENTESISABRE ParametrosDeEntradaDeUnaInstruccion PARENTESISCIERRA PUNTOYCOMA
123. ;
124.
125. InstruccionGirarALaDerecha::=
126.     DERECHA PARENTESISABRE ParametrosDeEntradaDeUnaInstruccion PARENTESISCIERRA PUNTOYCOMA
127. ;
128.
129. InstruccionTomarObjeto::=
130.     TOMAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
131. ;
132.
133. InstruccionSoltarObjeto::=
134.     SOLTAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
135. ;
136.
137. InstruccionEliminarObjeto::=
138.     ELIMINAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
139. ;
140.
141. InstruccionDesactivarKaboom::=
142.     DESACTIVARKABOOM PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
143. ;
144.
145. InstruccionPintarSuelo::=
146.     PINTAR PARENTESISABRE PaletaDeColores PARENTESISCIERRA PUNTOYCOMA
147. ;
148.
149. PaletaDeColores::=
150.     COLORES
151.     | /* Color predeterminado - No hay entrada de algún color. */
152. ;
153.
154. InstruccionDejarDePintarSuelo::=
155.     DEJAPINTAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
156. ;
157.
158. InstruccionImprimirVariables::=
159.     IMPRIMIRVARIABLE PARENTESISABRE IDENTIFICADOR PARENTESISCIERRA PUNTOYCOMA
160. ;
161.
162. InstruccionImprimirCadenas::=
163.     IMPRIMIRCADENA PARENTESISABRE Cadenas PARENTESISCIERRA PUNTOYCOMA
164. ;
165.
166. Cadenas::=
167.     IDENTIFICADOR ContinuacionDeCadenas
```

```
168.      | CADENA ContinuacionDeCadenas
169. ;
170.
171. ContinuacionDeCadenas::=
172.      OPERADORARITMETICOSUMA IDENTIFICADOR ContinuacionDeCadenas
173.      | OPERADORARITMETICOSUMA CADENA ContinuacionDeCadenas
174.      | /* Ningúna cadena más */
175. ;
176.
177. InstruccionTengoObjetoDelante::=
178.      OBJETODELANTE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
179. ;
180.
181. InstruccionTengoBombaDelante::=
182.      KABOOMDEFRENTE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
183. ;
184.
185. InstruccionQueTengoDelanteDeMi::=
186.      QUETENGODELANTE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
187. ;
188.
189. TengoMuroDelanteDeMi::=
190.      MURODELANTE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
191. ;
192.
193. InstruccionTerminarBloque::=
194.      TERMINARBLOQUE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
195. ;
196.
197. LlamadaAFuncion::=
198.      IDENTIFICADOR PARENTESISABRE Parametros PARENTESISCIERRA PUNTOYCOMA
199. ;
200.
201. Parametros::=
202.      NUMERO ParametrosContinuacion
203.      | IDENTIFICADOR ParametrosContinuacion
204.      | NumeroAleatorio ParametrosContinuacion
205.      | OperacionAritmetica ParametrosContinuacion
206.      | /* Puede no llevar identificadores */
207. ;
208.
209. ParametrosContinuacion::=
210.      SEPARADOR NUMERO ParametrosContinuacion
211.      | SEPARADOR IDENTIFICADOR ParametrosContinuacion
212.      | SEPARADOR NumeroAleatorio ParametrosContinuacion
213.      | SEPARADOR OperacionAritmetica ParametrosContinuacion
214.     | /* No envía más parámetros */
215. ;
216.
217. EstrucutraDeControlIf::=
218.      SI PARENTESISABRE Condicion PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones LLAVECIERRA
EstrucutraDeControlIfElse
219. ;
220.
221. Condicion::=
222.      CondicionDeComparacionSimple
223.      | CondicionLogica
224.      | CondicionDeComparacionConParentesis
225. ;
226.
227. CondicionDeComparacionSimple::=
228.      IDENTIFICADOR CondicionDeComparacionSimpleContinuacion
229.      | NUMERO CondicionDeComparacionSimpleContinuacion
230.      | NumeroAleatorio CondicionDeComparacionSimpleContinuacion
231.      | BOOLEANO
```

```
232. ;
233.
234. CondicionDeComparacionSimpleContinuacion::=
235.      OPERADORDECOMPARACION IDENTIFICADOR
236.      | OPERADORDECOMPARACION NUMERO
237.      | OPERADORDECOMPARACION NumeroAleatorio
238. ;
239.
240. CondicionDeComparacionConParentesis::=
241.      OperadorNegacion PARENTESISABRE CondicionDeComparacionSimple PARENTESISCIERRA
242. ;
243.
244. CondicionLogica::=
245.      CondicionDeComparacionConParentesis OPERADORLOGICO CondicionDeComparacionConParentesis
246. ;
247.
248. EstrucutraDeControlIfElse::=
249.      SINOSI PARENTESISABRE Condicion PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones
LLAVECIERRA EstrucutraDeControlIfElse
250.      | EstrucutraDeControlElse
251. ;
252.
253. EstrucutraDeControlElse::=
254.      SINO LLAVEABRE BloqueDeInstrucciones LLAVECIERRA
255.      | /* NO lleva nada*/
256. ;
257.
258. EstrucutraDeControlCase::=
259.      COMPARAR PARENTESISABRE IDENTIFICADOR PARENTESISCIERRA LLAVEABRE Casos CasoDefault
LLAVECIERRA
260. ;
261.
262. Casos::=
263.      CASO CondicionParaCasos DOSPUNTOS InstruccionesCasos
264.      | /* Sin más casos */
265. ;
266.
267. InstruccionesCasos::=
268.     BloqueDeInstrucciones FIN PUNTOYCOMA Casos
269.     | BloqueDeInstrucciones Casos
270. ;
271.
272. CondicionParaCasos::=
273.     OperadorDeComparacionOpcional NUMERO
274.     | OperadorDeComparacionOpcional NumeroAleatorio
275. ;
276.
277. OperadorDeComparacionOpcional::=
278.     OPERADORDECOMPARACION
279.     | /* No hay Operador de Comparación */
280. ;
281.
282. CasoDefault::=
283.      DEFAULT DOSPUNTOS BloqueDeInstrucciones FIN PUNTOYCOMA
284.      | DEFAULT DOSPUNTOS BloqueDeInstrucciones
285. ;
286.
287. EstrucutraDeControlFor::=
288.      PARA PARENTESISABRE DeclaracionesParaFor PUNTOYCOMA Condicion PUNTOYCOMA
InstruccionesParaFor PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones LLAVECIERRA
289. ;
290.
291. DeclaracionesParaFor::=
292.      IDENTIFICADOR AsignacionDeValoresAlDeclararVariable DeclaracionesParaForContinuacion
```

```
293.      | VARIABLE IDENTIFICADOR AsignacionDeValoresAlDeclararVariable
DeclaracionesParaForContinuacion
294.      | /* Puede ir nada */
295. ;
296.
297. DeclaracionesParaForContinuacion::=
298.      SEPARADOR IDENTIFICADOR AsignacionDeValoresAlDeclararVariable
DeclaracionesParaForContinuacion
299.      | SEPARADOR VARIABLE IDENTIFICADOR AsignacionDeValoresAlDeclararVariable
DeclaracionesParaForContinuacion
300.      | /* Ninguna otra declaración  */
301. ;
302.
303. InstruccionesParaFor::=
304.      IDENTIFICADOR OPERADORARITMETICOSUMA OPERADORARITMETICOSUMA
InstruccionesParaForContinuacion
305.      | IDENTIFICADOR OPERADORARITMETICORESTA OPERADORARITMETICORESTA
InstruccionesParaForContinuacion
306.      | /* Puede Ir nada */
307. ;
308.
309. InstruccionesParaForContinuacion::=
310.      SEPARADOR IDENTIFICADOR OPERADORARITMETICOSUMA OPERADORARITMETICOSUMA
InstruccionesParaForContinuacion
311.      | SEPARADOR IDENTIFICADOR OPERADORARITMETICORESTA OPERADORARITMETICORESTA
InstruccionesParaForContinuacion
312.      | /* Ninguna Otra Operacion */
313. ;
314.
315. EstrucutraDeControlDoWhile::=
316.      HACER LLAVEABRE BloqueDeInstrucciones LLAVECIERRA REPITE PARENTESISABRE Condicion
PARENTESISCIERRA PUNTOYCOMA
317. ;
318.
319. EstrucutraDeControlWhile::=
320.      REPITEHASTA PARENTESISABRE Condicion PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones
LLAVECIERRA
321. ;
322.
323. Funciones::=
324.      IDENTIFICADOR PARENTESISABRE ParametrosDeEntrada PARENTESISCIERRA LLAVEABRE
BloqueDeInstrucciones LLAVECIERRA Funciones
325.      | /* No hay funciones */
326. ;
327.
328. ParametrosDeEntrada::=
329.      VARIABLE IDENTIFICADOR MasParametrosDeEntrada
330.      | /* Sin parametros de entrada */
331. ;
332.
333. MasParametrosDeEntrada::=
334.      SEPARADOR VARIABLE IDENTIFICADOR MasParametrosDeEntrada
335.      | /* No más parámetros */
336. ;
337.
338. OperadorNegacion::=
339.      OPERADORNEGACION
340.      | /* No hay Operador*/
341. ;
342.
```

7.2 Modificaciones al proyecto

Para generar el Analizador Sintáctico es necesario realizar unas modificaciones al proyecto, las modificaciones realizadas se encuentran en el **Capítulo 5** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado.**

7.3 Pruebas del Analizador Sintáctico

A continuación, se muestran los resultados de las pruebas realizadas al Analizador Sintáctico una vez que se ha integrado al Entorno de Desarrollo.

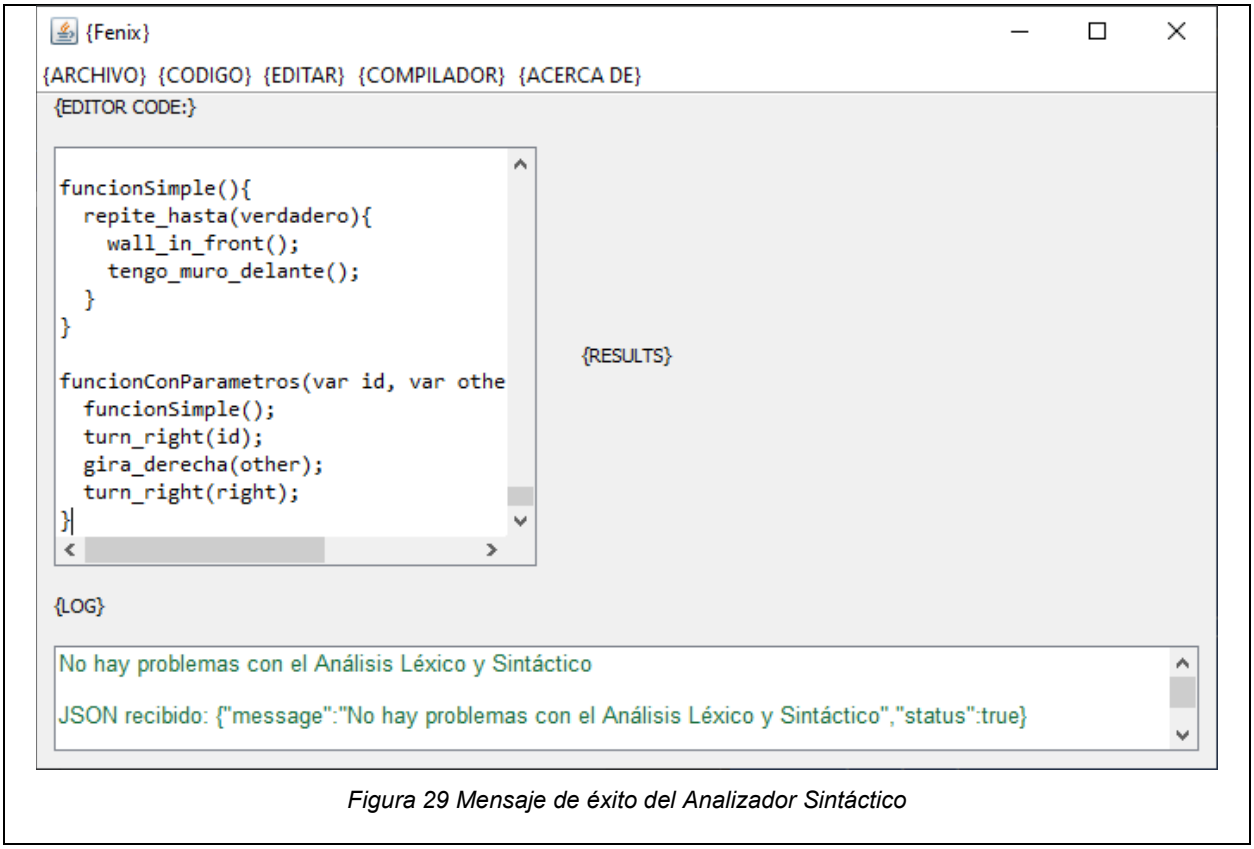

Figura 29 Mensaje de éxito del Analizador Sintáctico

Para este programa de prueba quiero aclarar las siguientes cosas:

- Se toman en cuenta casos extremos para probar la gramática. Pero así se han podido detectar errores de la gramática.
- Por si no se ha detectado, se declaran varias veces una misma variable. Para este caso de prueba se puede pasar desapercibido. En este punto se está probando solo la gramática, por lo que se puede ser altamente permisivos en ese aspecto.
- Hay algo que mencioné en la teoría que no se está tomando en cuenta por ahora, hablaré de ella en el próximo capítulo. Al lector de este texto: ¿logras detectar cuál es el elemento ausente?

Con esto se ha terminado esta parte del análisis léxico y sintáctico. Se puede concluir hasta ahora que ambos analizadores se conectan entre sí. En el próximo capítulo se detallarán aún más algunas partes del Analizador Sintáctico.

8 Generación de Código

Hasta ahora el compilador que se ha estado creando no genera código final. Hasta este momento sí lo se puede llamar compilador porque sí hace un análisis léxico y sintáctico.

Pero, ha llegado el momento de dotar al compilador la característica de generar código. Al final de este capítulo el compilador quedará terminado, y lo que resta es integrarlo a un programa final que lea el código objeto y lo ejecute.

8.1 El lenguaje objeto

Hasta ahora se ha estado trabajando un poco con JSON para lograr un pase de parámetros controlados entre clases. Por controlados me refiero a que se puede conocer qué información trae un objeto JSON. Es decir, se puede saber que el valor “status” de los objetos JSON que se ha creado, indican que es un dato booleano y significa que el programa se ejecutó con éxito o no (dependerá el caso).

Bueno, no me extenderé. El código objeto del compilador será un objeto JSON con las instrucciones del programa fuente. Esto quiere decir que el lenguaje que se ha creado se convertirá completamente a un objeto JSON.

8.1.1 ¿Por qué un objeto JSON?

Comenzaré referenciando un proyecto anterior a este.

En una primera versión de este proyecto, el Analizador Sintáctico iría generando instrucciones que posteriormente un programa final (la máquina virtual) tendría que leer y ejecutar. Este código se podía resolver parcialmente, al final, el código objeto dejaba pendientes algunos cálculos para algunas instrucciones que resultaban incompletas en el análisis.

Avanzar(),Mover(10),Mover(id),Eliminar(),Declara(variable),Operación(749/(id*78))
Ejemplo de cómo se veía el código objetivo en las primeras versiones de este proyecto

Las instrucciones como: avanzar en el tablero, declarar variables, y algunas operaciones básicas son fáciles de realizar, pues simplemente se leen y se convierten en un programa destino. El problema es cuando se comienza a tomar en cuenta las estructuras de control. Todavía recuerdo que doté al Analizador Sintáctico de ese proyecto la capacidad de resolver, incluso, las operaciones lógicas y de comparación.

El problema es que, por ejemplo, para una estructura if se tiene que repetir varias veces un bloque nuevo de código. Esto a lo mejor, se podía resolver con muchas pilas o incluso con estructuras de datos que guardaran esa información. Funcionaba parcialmente.

Más problemas iban surgiendo tratando de cubrir el alcance del proyecto, hasta que llegó el momento de la llamada a función. Estos problemas surgieron por el principio de que no le puedes indicar al Analizador Sintáctico que vaya a una línea específica (Que no sabes dónde se encuentra), validar los parámetros de entrada que son propios de una

función (Que sean los correctos, ni más ni menos), que lea las instrucciones que debería ejecutar y al terminar volver al punto dónde se inició. Además, que dentro de esos bloques de funciones existen subprogramas que son igualmente nuevas producciones de código (Como regresar a leer un nuevo programa escrito por el usuario).

La solución a lo anterior fueron los objetos JSON. Resulta que, para muchas API's en la red se aplican este tipo de objetos para pasar parámetros.

Cuando se llama a una API de alguna aplicación por lo general regresarán un objeto con la información precisa que se requieren, un ejemplo de esto es lo que regresa una API de Pokémon llamada PokéAPI. Esta API regresa precisamente un objeto JSON con información de un Pokémon del que se le consulte. Por ejemplo, se puede ingresar a esta dirección: https://pokeapi.co/api/v2/pokemon/pikachu regresará un JSON con la información de Pikachu. Entre ellos el tipo de Pokémon, e incluso refiere a otras direcciones de la API con más información del personaje. (Hay muchísima más información, pero sinceramente no las entiendo, no soy tan fan de Pokémon, lo siento).

Si se ingresa al enlace desde Firefox aparecerá una forma más visual para comprender esta cadena de texto, si se ingresa desde cualquier otro navegador seguramente se visualizará mucho texto. Aunque con algunas otras herramientas se puede visualizar mejor el objeto.

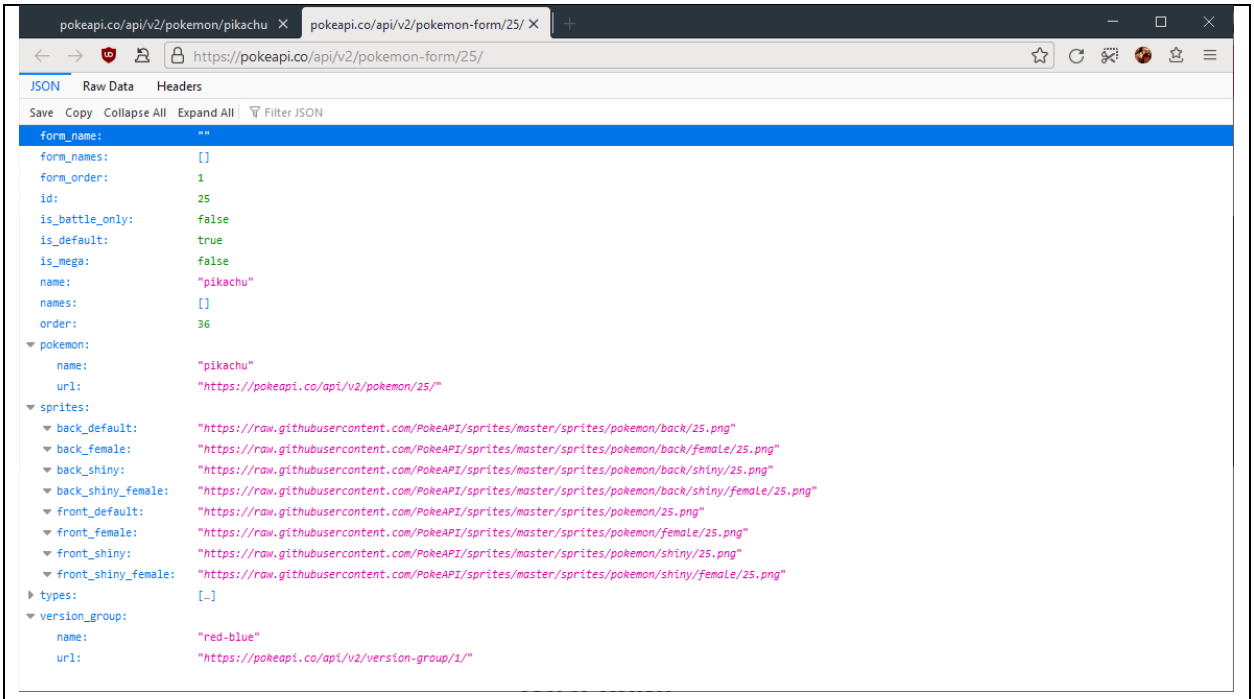

Figura 30 JSON en Firefox

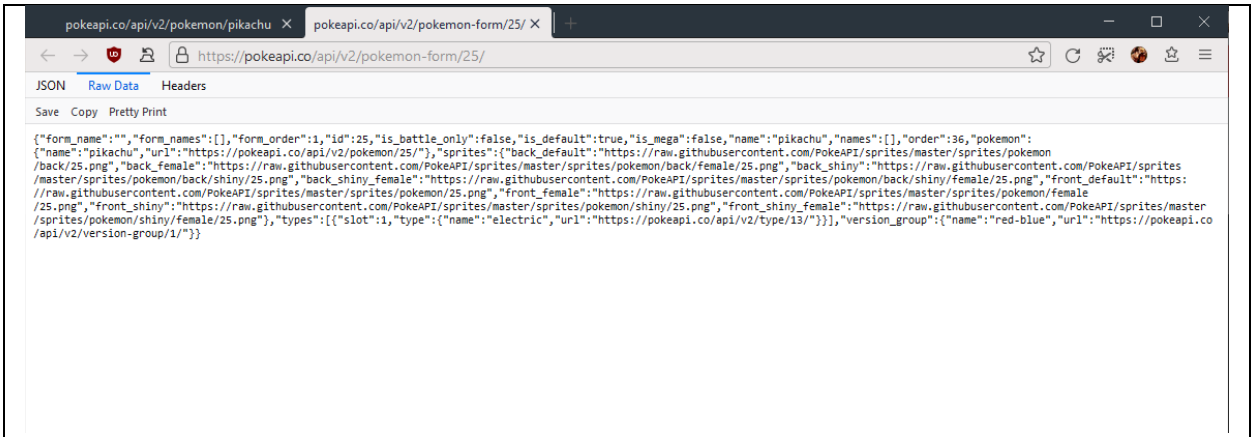

Figura 31 Cadena de texto de JSON

Bueno, justo estos objetos JSON dan la información necesaria de cualquier requerimiento que se tenga. Además, moverse por estos objetos es sumamente fácil. Es fácil ignorar bloques por completo. Leyendo solo algunas etiquetas se encontrará fácilmente lo requerido.

Y así fue como comencé a aplicarlo a este proyecto, comencé rehaciendo algunas producciones, realicé las pruebas y sorpresa: Fue más sencillo de leer la información que recién había analizado el Analizador Sintáctico, y además podía regresar más fácilmente a puntos específicos del programa fuente sin depender del Analizador Sintáctico.

Al final, terminé aplicando JSON para todo el proyecto. Cada programa fuente se convertía en un objeto JSON que finalmente otro programa tendría que interpretar y resolver. Hacer esto reduce mucho trabajo al Analizador Sintáctico pues este solo se dedica a generar código JSON mientras lee el programa fuente y ni siquiera se molestará en resolverlo.

Al final, el diagrama del compilador será el siguiente:

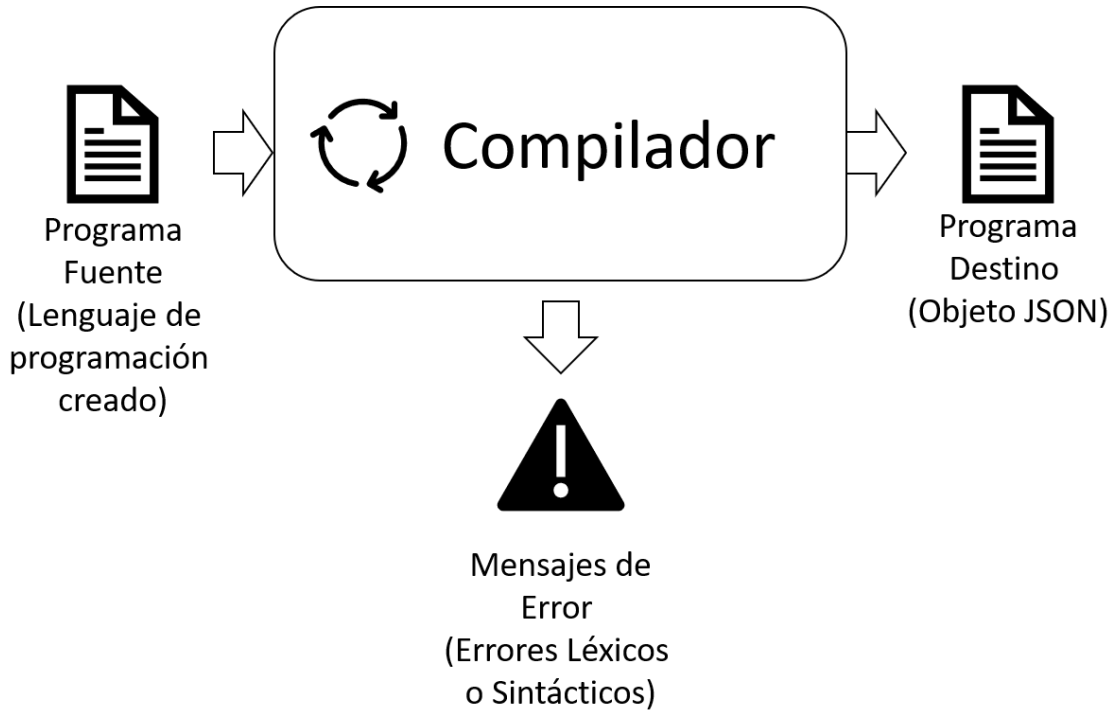

Figura 32 Diagrama del compilador hasta ahora. Imagen tomada de: (Aho, Lam, Sethi, & Ullman, 2008)

8.2 Estructura de CUP para generar código

Ahora que se ha puesto un objetivo de programa destino, comenzando a generar lo que pasará a ser el objeto JSON. En este capítulo todo el proyecto quedará igual con excepción del archivo "*Sintax.cup*" (Y las clases que se generen).

Pero antes, se debe tener en cuenta lo siguiente para comenzar a modificar el archivo de CUP que genera el Analizador Sintáctico. Estas reglas se basan en la documentación oficial de CUP (Hudson, n.d.)

1. En cualquier momento del análisis sintáctico se puede ejecutar código Java, para esto se utiliza de la siguiente estructura:

```
1.  NOMBRE ::=
2.  TERMINAL {: «CÓDIGO DEL USUARIO» :} NOTERMINAL TERMINAL NOTERMINAL {: «CÓDIGO DEL USUARIO» :}
3.  ;
4.
```

2. Se debe tener cuidado con ambigüedades que se puedan generar al intentar ejecutar código Java, ya que el Analizador Léxico no sabrá que código ejecutar primero y generará errores.

```
1.  NOMBRE ::=
2.  TERMINAL1 {: «CÓDIGO DEL USUARIO» :} NOTERMINAL1 TERMINAL2 NOTERMINAL3
3.  | TERMINAL1 {: «CÓDIGO DEL USUARIO» :} NOTERMINAL1 TERMINAL2 TERMINAL5 NOTERMINAL3
4.  ;
5.
```

En el ejemplo anterior, se observa una pequeña variación de la gramática, se nota que varía muy poco cada caso de la producción. Cuando el Analizador Sintáctico se encuentre en este punto de análisis, no sabrá cuál de los 2 códigos Java ejecutar. Aunque sea el mismo código Java el Analizador Sintáctico no compara si son iguales o no, por lo que, no sabrá cuál de ellos ejecutar y provocará un error, como consecuente el Analizador Sintáctico no puede continuar su análisis.

A continuación, pongo un ejemplo de cómo podría cambiar la gramática para evitar este tipo de conflicto, dependerá del creador de la gramática resolver estos errores.

```
1.  NOMBRE ::=
2.  TERMINAL1 {: «CÓDIGO DEL USUARIO» :} NOTERMINAL1 TERMINAL2 NUEVONOTERMINAL NOTERMINAL3
3.  ;
4.
5.  NUEVONOTERMINAL::=
6.  TERMINAL5
7.  | /* Nada */
8.  ;
9.
```

3. Las variables del código del usuario que se utilicen en un bloque no se comparten con otro bloque. Es decir, si se crea un objeto String en un Bloque 1, no se podrá usar en un Bloque 2. Cada bloque es distinto, para eso se pueden utilizar objetos globales que se pueden declarar al principio de la estructura de CUP.

```
1.  NOMBRE ::=
2.  TERMINAL {: «CÓDIGO DEL USUARIO 1» :} NOTERMINAL TERMINAL NOTERMINAL {: «CÓDIGO DEL USUARIO 2»
    :}
3.  ;
4.
```

4. Los Tokens (Terminales) pueden regresar el texto leído agregando un ":«Variable»" después de éste. A diferencia del caso anterior, estas variables se pueden utilizar en cualquier bloque de código Java.

```
1.  NOMBRE ::=
2.  TERMINAL:contenidoDelTerminal {: «CÓDIGO DEL USUARIO 1» :} NOTERMINAL TERMINAL NOTERMINAL {:
    «CÓDIGO DEL USUARIO 2» :}
3.  ;
4.
```

5. Las producciones de la gramática pueden regresar objetos con la instrucción RESULT, las producciones reciben estos objetos de la misma forma que el caso anterior.

```
1.   NOMBRE ::=
2.  TERMINAL {: «CÓDIGO DEL USUARIO 1» :} NOMBRE2:contenidoDelNoTerminal TERMINAL NOTERMINAL {:
    «CÓDIGO DEL USUARIO 2» :}
3.  ;
4.  NOMBRE2 ::=
5.  TERMINAL {: «CÓDIGO DEL USUARIO 1» :} NOTERMINAL TERMINAL NOTERMINAL:valorNoTerminal {: «RESULT
    = valorNoTerminal ; » :}
6.  | TERMINAL {: «CÓDIGO DEL USUARIO 1» :} NOTERMINAL:valorNoTerminal TERMINAL NOTERMINAL {:
    «RESULT = valorNoTerminal ; » :}
7.  ;
8.
```

8.3 Elementos clave del Analizador Sintáctico

Anteriormente cuando se genera el Analizador Sintáctico, escribí código en la parte de código de usuario. En este subtema detallaré el uso de cada uno de estos elementos.

Elemento:
private JSONObject resultadoJSON;
Descripción:
Es el objeto que contendrá el resultado final del Análisis, hay que inicializar este objeto cuando comience la lectura del Analizador Sintáctico.
Elemento:
private Stack pilaDeInstrucciones = new Stack(); private Stack pilaDePilas = new Stack();
Descripción:
Estas pilas ayudarán demasiado para guardar partes del código que podrán ser de ayuda después. Debido a la facilidad y versatilidad para utilizarlos se convertirán en objetos clave.

Elemento:
private ArrayList<JSONObject> instruccionesDelPrograma = new ArrayList<>();
Descripción:
Es una lista de objetos que albergará los objetos JSON que son las instrucciones que se van leyendo.

Elemento:

public JSONObject getResultadoJSON() {
return resultadoJSON;
}

Descripción:
Debido a que el objeto del resultado final del análisis sintáctico es privado, se crea un método getter para que elementos del otro programa puedan consultarlo.

Elemento:

public Symbol getSimbolo(){
return this.simbolo;
}

Descripción:
Es un método que regresa el símbolo que se está analizando para que se pueda utilizar en otras partes del proyecto.

8.4 Documentación de los objetos JSON de cada instrucción

En este capítulo no detallaré cómo realizar la generación de código para el análisis sintáctico porque no acabaría (Razón por la cual fue el subtema anterior). En cambio, solo agregaré las instrucciones que tendrá que generar el Analizador Sintáctico.

8.4.1 Estructura básica de un programa

La estructura básica del programa es que siempre se agregue la cadena “programa(){}”. Esto definirá en primera instancia que el lenguaje que se creó está siendo bien utilizado. Para esto se modifica la producción inicial de la gramática para que regrese los siguientes valores:

Código de Ejemplo del lenguaje:

```
1.  programa(){
2.  }
```

JSON de respuesta:

```
1.  {
2.  "functions":[],
3.  "main":[],
4.  "message":"success",
5.  "status":true
6.  }
```

Explicación de la estructura JSON:

status: Indica que el programa ha sido generado con éxito.
message: Indica un mensaje de éxito.
main: Contiene un arreglo de estructuras con las instrucciones del programa principal, será la primera estructura que se lea.
functions: Contiene un arreglo de estructuras dónde se encuentran definidas las distintas funciones disponibles del programa.

El resto de documentación se encuentra en el **Apéndice 3: Documentación del Código Destino.**

8.5 Compilador en pruebas

Una vez que se ha terminado de modificar los archivos originales del Analizador Sintáctico, se tendrá que probar y asegurarse que el programa regresa efectivamente un objeto JSON.

Para realizar esto, se propone ejecutar el siguiente programa de prueba, se puede descargar desde esta dirección web debido a que este programa de prueba consta de más de 327 líneas.

Abriremos el Entorno de Desarrollo y ejecutaremos el programa fuente. Observaremos que lo ha ejecutado correctamente.

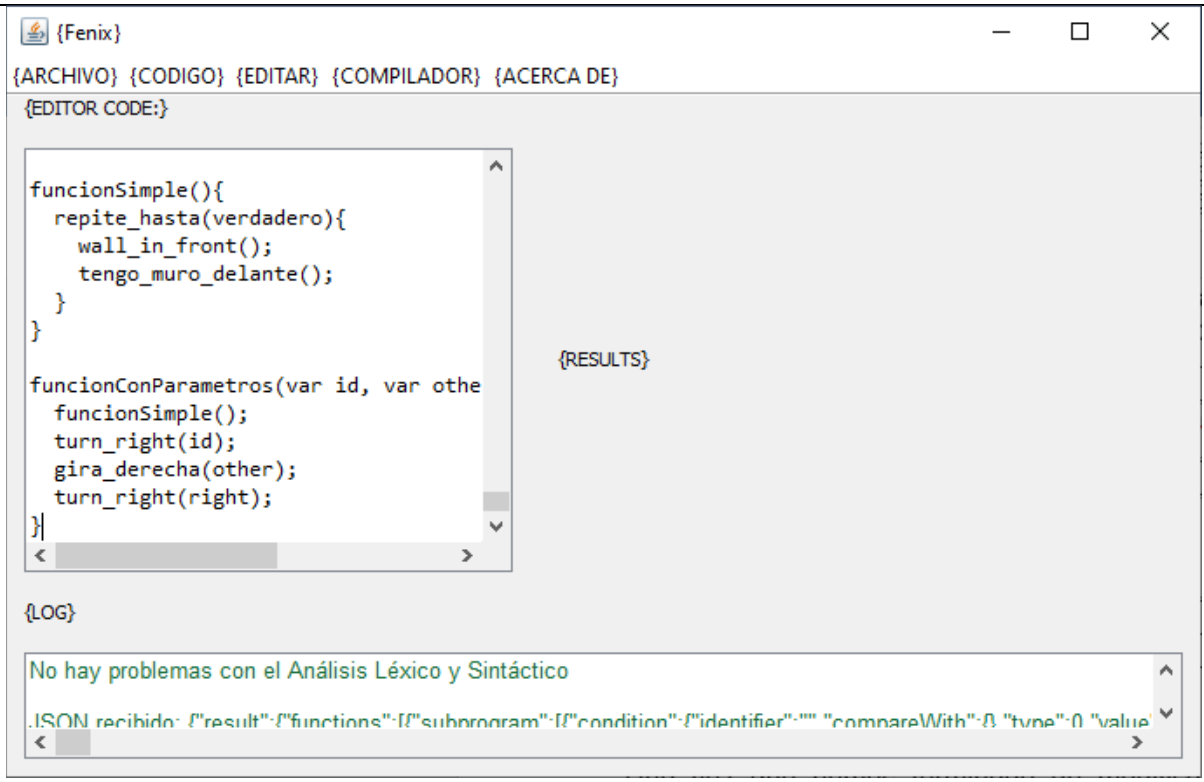

Figura 33 Mensaje de éxito del análisis del compilador

Para consultar el JSON resultante de este programa de prueba se puede consultar la siguiente dirección web. No se publica aquí porque tiene más de 4600 líneas de código.

8.6 Tabla de Símbolos

Al final del capítulo anterior mencioné que a este compilador le ha faltado algo. Bueno, este elemento es la tabla de símbolos. Mencioné al principio de este trabajo escrito, que ambos analizadores se estarían comunicando constantemente con la Tabla de Símbolos,

y en ese momento parecía un elemento sumamente importante. En estos párrafos quiero explicar por qué se puede omitir este elemento del compilador.

El compilador solo traduce un lenguaje a otro, no resuelve absolutamente ninguna operación o instrucción. ¿debería hacerlo? Sí. Sin embargo, como mencioné al principio de este capítulo, para resolver algunas cosas que requieren de analizar el código de otras partes del programa fuente, como ya expliqué, no se le puede decir al Analizador Sintáctico algo como: “Oye, me han pedido ejecutar una función (o una estructura iterativa), ve a traer las instrucciones que tiene esa parte del código porque urge ejecutarlas, valida esa función y si está correcta regresas con las instrucciones ya resueltas”. ¿Y si pide el usuario realizar una iteración? Tampoco se puede estar leyendo el código fuente una y otra vez. El proceso de lectura de código fuente únicamente se realiza una vez.

Sí podría haber una manera de hacerlo en el sintáctico como, se me ocurre, declarar las funciones al principio analizarlas e ir guardando ese análisis en una estructura de datos que lleva el nombre de Tabla de Símbolos. Así mismo se tiene que estar guardando mucha información en esa tabla conforme se vaya leyendo el código fuente.

Pero hay varios problemas que pueden traer este análisis, las cuales son:

- Se estarán haciendo las mismas estructuras JSON. Y la razón es que JSON ha permitido con su versatilidad poder moverse libremente y generar metadatos del código fuente leído. Por ejemplo, en una instrucción if: Si no se cumple la función simplemente se puede omitir el bloque de código tan fácil y sencillo. Si fuera el Analizador Sintáctico el encargado de realizar esta tarea se tendría que ir validando condiciones, o terminando de resolver operaciones aritméticas. Todo esto conforme se va leyendo y no, no es la manera de hacerlo.
- Complicará mucho más el proyecto. El lenguaje que se ha propuesto abarca muchas cosas de un lenguaje de alto nivel y aún, le hace falta muchas características. El código para generar el Analizador Sintáctico es grande sin instrucciones (más de 450 líneas de código), quedó aún más grande cuando se genera código (más de 1450 líneas de código). Al lector le pregunto: ¿Se imaginan de cuántas líneas hubiera quedado si todavía le se hubiera agregado código para resolver cosas?
- El proyecto como está podría validar, al menos, que las variables no se repitan o que hayan sido declaradas previamente. Es decir, que si el Analizador Sintáctico se encuentra con la instrucción de modificar una variable tendría que verificar que, al menos, esta haya sido declarada. O que si el analizador se encuentra que tiene que declarar 2 veces (o más) una variable con el mismo nombre debería detenerse y decir algo como: “Esto está mal, estás declarando varias veces la misma variable”. Pero hacer esto solo agregará código de más y solo se estará validando una cosa de las tantas que aún hay por hacer.
- Portabilidad. El diseño que se ha generado hasta ahora del compilador permite una independencia del compilador con cualquier otro componente, es decir, no

depende de otros componentes. Como mencioné anteriormente, el compilador no interactuará con ninguna otra clase más que con los analizadores mismos. Gracias a esto, el compilador puede utilizarse en otro proyecto, por ejemplo, se puede llevar a web.

- Las instrucciones se deben resolver en el momento. Recordando que el objetivo del lenguaje: es mover piezas en un tablero. Estas piezas se tienen que mover conforme el programa principal lo indique, no se puede esperar a resolver para después.
- Lo último me lleva a otra contra: el programa se debe comunicar con otros elementos del Entorno de Desarrollo. El tablero es un componente del Entorno de Desarrollo, no es algo que genere el compilador. Comunicarse con este tablero dificultará aún más el código del Analizador Sintáctico.

Por estas razones se ha desechado la característica del compilador de tener una tabla de símbolos.

8.7 Terminando el compilador

Con esta última prueba se ha terminado por completo el compilador. Y hasta aquí podría quedar esta tesis. Pero no sucederá así. Ahora se tiene que hacer algo con este código fuente en formato JSON.

Es lo último que falta, es decir, integrar este compilador a un proyecto Entorno de Desarrollo y muestre los resultados.

9 Compilador bilingüe

He dicho muy poco sobre el soporte de idiomas del compilador. Pero para llegar a este punto era necesario llegar a generar un compilador estable y funcional, pero, sobre todo que el compilador se encuentre en una etapa final dónde casi no requerirá de modificaciones. Podría decirse que el compilador ha entrado ahora a lo que se conoce en la ingeniería de software como “etapa de mantenimiento”. Ya se ha terminado con él, sin embargo, aún se podría agregar características o modificar ciertos componentes para mejorarlos, pero estos serían cambios menores. El trabajo pesado ha terminado.

En este capítulo, se comenzará a reciclar código que se ha estado realizando con el objetivo de generar 2 compiladores más.

Las instrucciones del compilador las reconoce indistintamente si están en inglés o español, pero ¿qué sucede si solo quiero utilizar instrucciones únicamente en inglés o únicamente en español? Es decir, que el compilador sea capaz de reconocer un error cuando se tiene configurado el compilador como “solo español” cuando una instrucción está en inglés.

El Analizador Léxico es el encargado de separar el programa fuente en tokens, gracias a las expresiones regulares que se ha creado. Si somos más observadores, estas expresiones regulares están diseñadas para reconocer ambos idiomas. Entonces lo que se realizará, será quitarle esa característica para que solo reconozca instrucciones para el idioma que se requiere.

Ahora retomando al Analizador Sintáctico, este se comunica constantemente con el Analizador Léxico. No hay forma de indicarle a un Analizador Sintáctico para que solo utilice cierto Analizador Léxico a conveniencia del usuario. Por lo que se tendrá que generar una vez más.

En ambos casos, son actualizaciones menores porque ya está terminado todo.

9.1 Multi-compilador

Tomando en cuenta lo anterior, se tienen que modificar clases ya realizadas para poder tener un multi soporte de idiomas. Las modificaciones realizadas se encuentran detalladas en el **Capítulo 6** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado.**

9.2 Pruebas de idioma al compilador

Para probar que se están haciendo las cosas de manera correcta, se tendrá que modificar de momento la instrucción que llama a estas instrucciones modificando el parámetro de lenguaje español, inglés o default según sea el caso.

```
1.  resultado = Analizadores.EjecutarCompilador(leerCodigoFuenteDelUsuario(),"ES");
```

El resultado del programa (si se modifica a “solo español”) tendrá que dejar de reconocer las instrucciones en inglés.

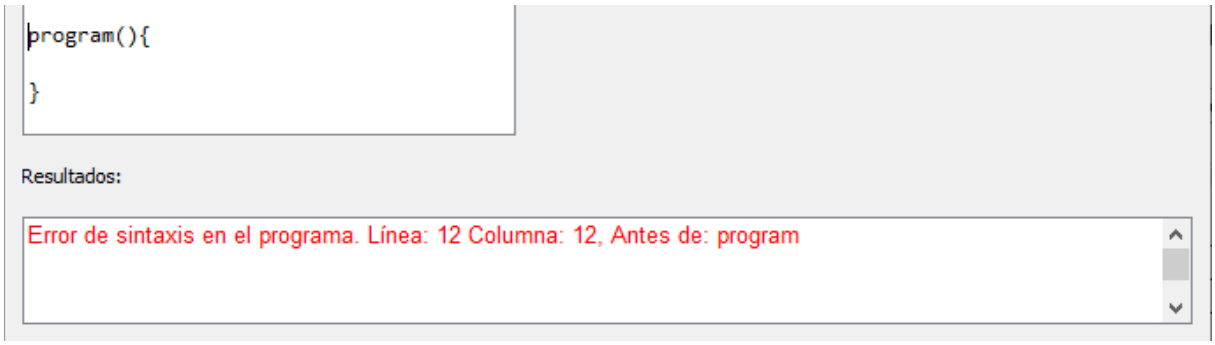

Figura 34 Programa con error

Con esto ha quedado ese soporte de idiomas del compilador. Aunque son 3 compiladores distintos, responden al diseño original y a los objetivos del proyecto.

10 Soporte de Idiomas

Uno de los objetivos del IDE es el soporte de idiomas en inglés y español. En este capítulo dotaré al IDE la capacidad de cambiar de idioma según prefiera el usuario final.

La idea de este capítulo es que se pueda configurar un lenguaje de usuario preferido, todas las instrucciones deberán mostrarse en inglés o español según sea el caso. Para lograr esto, se volverá a trabajar con los formatos JSON.

Para esto se requerirá de una clase nueva. Esta clase leerá archivos de texto con cadenas de texto en distintos idiomas.

Para conocer el proceso y documentación sobre el soporte de idiomas, se puede consultar el **Capítulo 7** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado.**

11 Mapa virtual

Antes de continuar es necesario definir cómo se va a leer un mapa personalizado en el IDE, este será el primer paso necesario para la creación de la máquina virtual.

Cargar un mapa en el IDE es necesario para el funcionamiento de la máquina virtual porque la máquina virtual comenzará a mover el personaje conforme las instrucciones lo vayan indicando.

El concepto de como se ve un mapa ya se ha hablado antes en el *Capítulo 4.4.1: El Mapa*, ahora es el turno de representar cómo se va a leer un mapa en el programa.

11.1 Valores separados por comas CSV

Una solución a este problema es utilizar los valores separados por comas para la creación de estos mapas.

Esto ofrece una mayor flexibilidad para la creación de diferentes mapas con distintas situaciones a resolver. Solo bastará con abrir una hoja de cálculo, introducir algunos valores que reconozca el IDE y el mapa se deberá mostrar sin inconvenientes.

La forma en la que se está pensando es la siguiente:

Una clase que lea un archivo CSV, que genere un mapa virtual para que la máquina virtual trabaje sobre él y que genere un mapa visual para el usuario.

11.1.1 Definición del archivo

Primero se definen los símbolos que tendrá un archivo CSV. Para generar un mapa se requiere obligatoriamente el siguiente símbolo:

- La posición inicial del personaje. El personaje debe estar en el mapa, solo 1 vez puede aparecer y puede estar mirando al Norte, Sur, Este u Oeste. Este se representa con las letras N, S, E, W respectivamente.

Los siguientes símbolos no son obligatorios, sin embargo, a veces son necesarios para que aparezcan correctamente en el archivo generado. Si no se definen los objetos que se muestran a continuación, podría provocar que los elementos del mapa no aparezcan correctamente.

- Un símbolo ‘0’ se ignora por completo, pero ayudará a delimitar el tamaño del mapa del personaje.
- El símbolo ‘;’ va al final del mapa y es opcional. Simplemente ayuda a delimitar el mapa.

Por ejemplo, si se desea crear un mapa simple únicamente con el personaje en una posición, se puede abrir una hoja de cálculo, colocar el símbolo ‘w’ indicando que el personaje debe aparecer mirando al Oeste. Dependiendo de la hoja de cálculo que se utilizarán es posible que en el CSV se pierda la posición dónde se coloca el personaje y, además, no se ha puesto un límite del mapa. Es decir, no se sabrá dónde inicia el mapa o dónde termina.

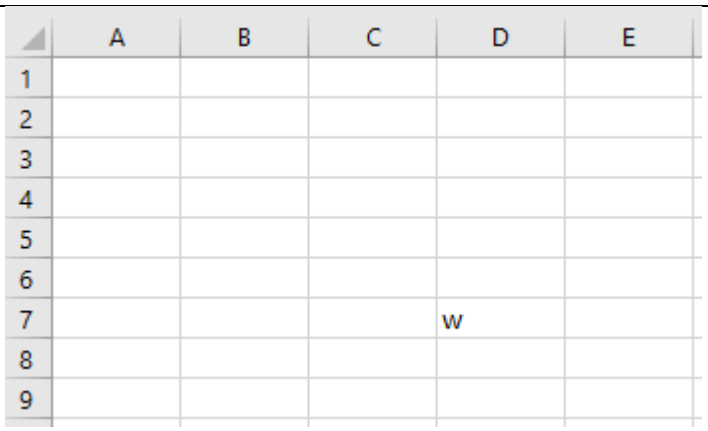

Figura 35 Hoja de cálculo

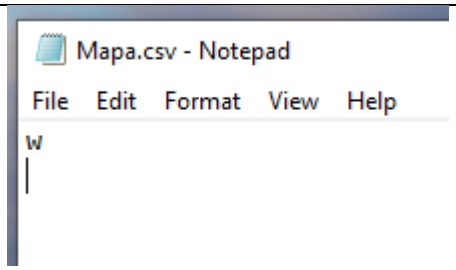

Figura 36 Contenido erróneo de un archivo CSV

Para resolver esto se puede utilizar los símbolos ‘0’ y/o ‘;’, al guardar la hoja de cálculo como CSV aparecerán todas las áreas que se delimitan.

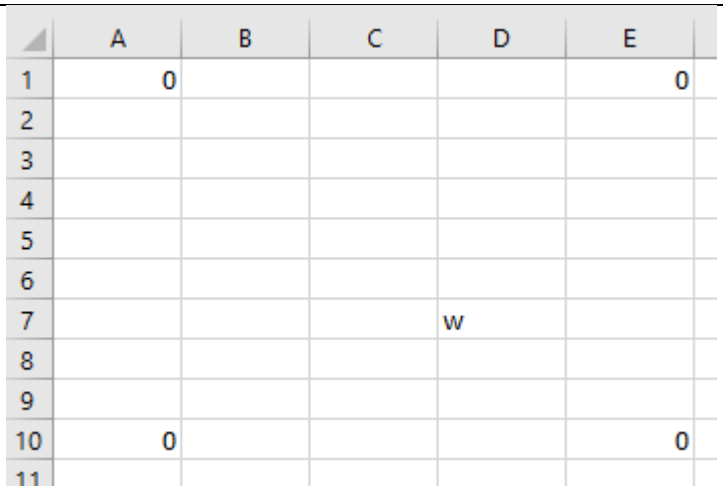

Figura 37 Hoja de cálculo con límites fijos

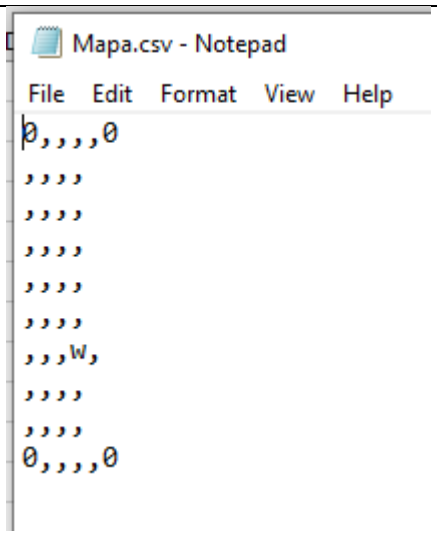

Figura 38 Archivo CSV correcto

Gracias a esos símbolos se puede crear el mapa simple como se desee.

Los siguientes símbolos ayudarán a crear mapas más dinámicos con obstáculos. Estos símbolos pueden aparecer más de una vez.

- El símbolo de Meta: El símbolo de meta es una bandera a dónde el personaje podría terminar el juego. Solo emite un mensaje de éxito cuando termina las instrucciones y se encuentra sobre el símbolo de meta. Se representa por la letra Q.
- Símbolo de Bomba: El símbolo de bomba es un obstáculo que, si no es desactivado, el juego se detiene y no se podrá continuar con el juego. Se representa por la letra K.
- Símbolo de Objeto: Este símbolo es un Objeto que el personaje puede recoger y es en lo que se convierte cuando una bomba es desactivada. El personaje solo puede recoger un objeto a la vez. Se representa por la letra O.
- Objeto no traspasable. Este símbolo representa un objeto que no tiene interacción con el usuario. Se representa por la letra P.

Se creará algo similar a esto:

	A	B	C	D	E	F	G	H
1	p	p	p	p	p	p	p	p
2	p	q	k	k	k	k	q	p
3	p	k	o	o	o	o	k	p
4	p	k	o			o	k	p
5	p	k	o			o	k	p
6	p	k	o			o	k	p
7	p	k	o	w		o	k	p
8	p	k	o			o	k	p
9	p	k	o			o	k	p
10	p	k	o			o	k	p
11	p	k	o			o	k	p
12	p	k	o			o	k	p
13	p	k	o	o	o	o	k	p
14	p	q	k	k	k	k	q	p
15	p	p	p	p	p	p	p	p

Figura 39 Hoja de Cálculo

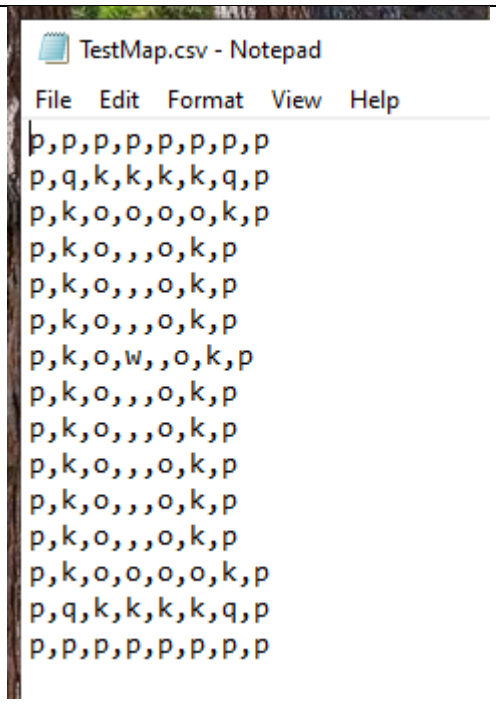

Figura 40 Archivo CSV

11.2 Generando un mapa

El siguiente paso es dotar al IDE la capacidad de generar este mapa. Para esto, se buscará una forma de representar el mapa tanto de forma virtual como de forma gráfica.

Para representarlo de forma virtual se utilizará un simple arreglo bidimensional, este arreglo bidimensional tendrá cada una de las posiciones del mapa. La forma de representar este mapa virtual será a través de números enteros. Cada número entero representará un objeto del mapa.

Para generar el mapa se creará una clase que se llamará "*CargaDeMapaCSV*", esta clase será la encargada de leer la cadena de texto CSV. Esta clase tendrá algunos objetos globales las cuales son:

```
1.      private int[][] MapaBackend;
2.      private int Filas;
3.      private int Columnas;
4.      private String MapaHTML;
5.      private boolean status;
6.      private String message;
7.
```

Dónde:

- El objeto MapaBackend será el mapa virtual de números enteros.
- El objeto Filas y Columnas serán las filas y columnas del mapa virtual.
- El objeto MapaHTML será el mapa en su versión gráfica.
- El objeto status será un indicador que el mapa ha sido leído de forma correcta.
- El objeto message será una cadena de texto con un mensaje que se genera según el caso de éxito o de error.

Lo primero que hay que realizar, es obtener el tamaño de columnas y filas totales a partir del archivo CSV, este se logra con un algoritmo simple que cuente las comas y los saltos de línea. Todos estos datos se estarán guardando en un objeto global de Filas y Columnas que después será consultado por la máquina virtual.

Esto se logra gracias a un sencillo algoritmo:

```
1.          Filas=1;
2.          Columnas=1;
3.          for(int i=0; i<CSVMapa.length() ; i++){
4.              if(CSVMapa.charAt(i) == ','){
5.                  Columnas += 1;
6.              }else if(CSVMapa.charAt(i) == '\n'){
7.                  Filas+=1;
8.                  Columnas = 1;
9.              }
10.         }
11.
```

Para finalizar, se requiere de un algoritmo que sea capaz de leer y colocar en las celdas correctas del mapa virtual las posiciones de objetos.

La idea de es colocar números que representarán los objetos en el mapa. Estos números significarán lo siguiente:

- El número 0: Indica una casilla vacía.
- El número 1: Indicará que el personaje está mirando al norte.
- El número 2: Indicará que el personaje está mirando al sur.
- El número 3: Indicará que el personaje está mirando al este.
- El número 4: Indicará que el personaje está mirando al oeste.
- El número 5: Indicará que se trata de una casilla meta.
- El número 6: Indicará que se trata de una casilla con una bomba.
- El número 7: Indicará que se trata de una casilla con un objeto.
- El número 8: Indica que se trata de una casilla que no es accesible al usuario.

Además, se debe tener en cuenta que:

- Se ignora el símbolo “0” del archivo CSV.
- Cada salto de línea es un indicador del fin de fila.
- Se debe considerar que si se encuentra un “;” es un indicador del final del análisis.

Esto se logra con un sencillo algoritmo:

```
1.          int jFilas=0;
2.          int kColumnas=0;
3.          for(int i=0; i<CSVMapa.length() ; i++){
4.              switch(CSVMapa.charAt(i)){
5.                  case 'N': case 'n':
6.                      if(!personajeLeido){
7.                          MapaBackend[jFilas][kColumnas] = 1;
8.                      }else{
9.                          this.status=false;
10.                         this.message=idioma.traerTexto("ERRORPERSONAJE");
11.                         return;
12.                     }
13.                     break;
14.                 case 'S': case 's':
15.                     if(!personajeLeido){
16.                         MapaBackend[jFilas][kColumnas] = 2;
17.                     }else{
18.                         this.status=false;
19.                         this.message=idioma.traerTexto("ERRORPERSONAJE");
20.                         return;
21.                     }
22.                     break;
23.                 case 'E': case 'e':
24.                     if(!personajeLeido){
25.                         MapaBackend[jFilas][kColumnas] = 3;
26.                     }else{
27.                         this.status=false;
28.                         this.message=idioma.traerTexto("ERRORPERSONAJE");
29.                         return;
30.                     }
31.                     break;
32.                 case 'W': case 'w':
33.                     if(!personajeLeido){
34.                         MapaBackend[jFilas][kColumnas] = 4;
35.                     }else{
```

```
                        this.status=false;
                        this.message=idioma.traerTexto("ERRORPERSONAJE");
                        return;
                    }
                    break;
                case 'X': case 'x':
                    MapaBackend[jFilas][kColumnas] = 5;
                    break;
                case 'K': case 'k':
                    MapaBackend[jFilas][kColumnas] = 6;
                    break;
                case 'O': case 'o':
                    MapaBackend[jFilas][kColumnas] = 7;
                    break;
                case 'P': case 'p':
                    MapaBackend[jFilas][kColumnas] = 8;
                    break;
                case ',':
                    kColumnas++;
                    break;
                case '\n':
                    jFilas++;
                    kColumnas = 0;
                    break;
                case ';':
                    generarMapaHTML();
                    status=true;
                    message=idioma.traerTexto("MAPACREADO");
                    return;
                case '0':
                    break;
                default:
                    status=false;
                    message=idioma.traerTexto("ERRORSIMBOLOS");
                    return;
            }
        }
```

El siguiente algoritmo a desarrollar es la representación del mapa en forma gráfica. Una forma que el usuario sea capaz de visualizar con la vista.

Para esta sección se deberá tomar en cuenta el IDE que se crea. Existe un elemento que se crea a la hora de diseñar, se trata de un objeto *JLabel* llamado "*lblResultados*". En este objeto es dónde el usuario visualizará el mapa gráfico.

La forma en la que se me ocurrió mostrar los resultados es a través de texto y emojis. Es una solución temporal, pero funciona. Para poder colocar el mapa en un objeto *label*, es necesario considerar una cosa: Para poder soportar multilíneas, el texto debe estar en HTML con las etiquetas necesarias para generarlo.

Es decir, que ahora se tendrá que generar HTML para mostrar el mapa que se acaba de leer. Ahora se tiene que pensar en una solución sobre cómo representar los objetos del mapa virtual.

Tomando en cuenta que se requiere que tenga un color de fondo cuando se requiera, se pondrán los emojis dentro de una tabla que cambie de color de celda cuando sea necesario. La estructura quedará de la siguiente forma:

```
1.  <html>
2.        <body><br>
3.                  <table>
4.                            <tr>
5.                                      <td style="background-color:# «COLOR DE FONDO EN
    HEXADECIMAL»;"> «EMOJI» </td>
6.                                      <td style="background-color:# «COLOR DE FONDO»;"> «EMOJI»
    </td>
7.                                      <td style="background-color:# «COLOR DE FONDO»;"> «EMOJI»
    </td>
8.                            </tr>
9.
10.                 </table><br>
11.
12.       </body>
13. </html>
14.
```

Gracias a lo anterior, las filas y celdas que se crean no tienen límite. Salvo el límite del espacio. Así se pueden crear programas más dinámicos.

Para generar el texto en HTML se ocupará un sencillo algoritmo el cual es el siguiente:

```
1.      private void generarMapaHTML(){
2.          for(int i = 0; i<Filas ; i++ ){
3.              MapaHTML += "<tr>";
4.              for(int j = 0; j<Columnas ; j++ ){
5.                  switch(this.MapaBackend[i][j]){
6.                      case 0:
7.                          MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">□</td>";
8.                          break;
9.                      case 1:
10.                         MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">⬆</td>";
11.                         break;
12.                     case 2:
13.                         MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">⬇</td>";
14.                         break;
15.                     case 3:
16.                         MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">➡</td>";
17.                         break;
18.                     case 4:
19.                         MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">⬅</td>";
20.                         break;
21.                     case 5:
22.                         MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">🏁</td>";
23.                         break;
24.                     case 6:
25.                         MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">💣</td>";
26.                         break;
27.                     case 7:
28.                         MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">🚪</td>";
29.                         break;
30.                     case 8:
```

```
                        MapaHTML += "<td style=\"background-color:"+idioma.traerColor("DEFAULT")
    +";\">✕</td>";
                        break;
                    default:

                }
            }
            MapaHTML += "</tr>";
        }
        MapaHTML = "<html><body><br><table>" + MapaHTML + "</table><br></body></html>";
    }
```

11.3 Visualizando el mapa

Ahora ha llegado el momento de testear estos algoritmos, para ello se programará el botón "*Abrir mapa*". Se ocupará un selector de archivos de la máquina virtual de Java.

Se programará el objeto "*abrirCodigo*". El código es el siguiente:

```
        abrirCodigo.addActionListener(e -> {
            if(!ejecutarPrograma.getCanStart()) {
                accionInvalida();
                return;
            }
            JFileChooser exploradorDeArchivos = new JFileChooser();
            int response = exploradorDeArchivos.showOpenDialog(null);
            if(response == JFileChooser.APPROVE_OPTION){
                abrirProgramaSeleccionadoUsuario(new
    File(exploradorDeArchivos.getSelectedFile().getAbsolutePath()));
            }
        });
```

Después se programará el método "*abrirMapaSeleccionadoUsuario*". Este método será el encargado de leer el archivo seleccionado por el usuario y llamar al método para generar el mapa virtual y visual inicial.

El algoritmo para leer el archivo seleccionado es el siguiente:

```
        String string = "";
        if(ruta!=null){
            rutaMapaDelUsuario = ruta.toString().replace("\\", "/");
        }
        String filePath = rutaMapaDelUsuario;
        try {
            List<String> lines = Files.readAllLines(Paths.get(filePath),
    StandardCharsets.UTF_8);
            string = String.join("\n", lines);
        } catch (IOException ignored) {}
```

Inmediatamente después, se genera el mapa virtual y visual.

```
        CargaDeMapaCSV map = new CargaDeMapaCSV();
        map.generarMapa(string);
```

Si el status es correcto, lo único que falta es colocar el texto en HTML que se generó en el *JLabel* "*lblResultados*" y agregar un texto con el mensaje de éxito. De lo contrario se tendrá que indicar al usuario el mensaje de error generado.

```
1.          if(map.isStatus()){
2.              lblResultados.setText(map.getMapaHTML());
3.              agregarTexto(map.getMessage());
4.          }else{
5.              agregarTextoError(map.getMessage());
6.          }
```

En el caso del ejemplo que se ha estado trabajando desde el *Capítulo 11.1.1: Definición del archivo* quedará el código HTML así:

```
1.  <html>
2.     <body>
3.        <br>
4.        <table>
5.           <tr>
6.              <td style="background-color:#BDB76B;">❌</td>
7.              <td style="background-color:#BDB76B;">❌</td>
8.              <td style="background-color:#BDB76B;">❌</td>
9.              <td style="background-color:#BDB76B;">❌</td>
10.             <td style="background-color:#BDB76B;">❌</td>
11.             <td style="background-color:#BDB76B;">❌</td>
12.             <td style="background-color:#BDB76B;">❌</td>
13.             <td style="background-color:#BDB76B;">❌</td>
14.          </tr>
15.          <tr>
16.             <td style="background-color:#BDB76B;">❌</td>
17.             <td style="background-color:#BDB76B;">🏁</td>
18.             <td style="background-color:#BDB76B;">💣</td>
19.             <td style="background-color:#BDB76B;">💣</td>
20.             <td style="background-color:#BDB76B;">💣</td>
21.             <td style="background-color:#BDB76B;">💣</td>
22.             <td style="background-color:#BDB76B;">🏁</td>
23.             <td style="background-color:#BDB76B;">❌</td>
24.          </tr>
25.          <tr>
26.             <td style="background-color:#BDB76B;">❌</td>
27.             <td style="background-color:#BDB76B;">💣</td>
28.             <td style="background-color:#BDB76B;">🗋</td>
29.             <td style="background-color:#BDB76B;">🗋</td>
30.             <td style="background-color:#BDB76B;">🗋</td>
31.             <td style="background-color:#BDB76B;">🗋</td>
32.             <td style="background-color:#BDB76B;">💣</td>
33.             <td style="background-color:#BDB76B;">❌</td>
34.          </tr>
35.          <tr>
36.             <td style="background-color:#BDB76B;">❌</td>
37.             <td style="background-color:#BDB76B;">💣</td>
38.             <td style="background-color:#BDB76B;">🗋</td>
39.             <td style="background-color:#BDB76B;">□</td>
40.             <td style="background-color:#BDB76B;">□</td>
41.             <td style="background-color:#BDB76B;">🗋</td>
42.             <td style="background-color:#BDB76B;">💣</td>
```

```
43.            <td style="background-color:#BDB76B;">❌</td>
44.         </tr>
45.         <tr>
46.            <td style="background-color:#BDB76B;">❌</td>
47.            <td style="background-color:#BDB76B;">💣</td>
48.            <td style="background-color:#BDB76B;">🏳</td>
49.            <td style="background-color:#BDB76B;">⬜</td>
50.            <td style="background-color:#BDB76B;">⬜</td>
51.            <td style="background-color:#BDB76B;">🏳</td>
52.            <td style="background-color:#BDB76B;">💣</td>
53.            <td style="background-color:#BDB76B;">❌</td>
54.         </tr>
55.         <tr>
56.            <td style="background-color:#BDB76B;">❌</td>
57.            <td style="background-color:#BDB76B;">💣</td>
58.            <td style="background-color:#BDB76B;">🏳</td>
59.            <td style="background-color:#BDB76B;">⬜</td>
60.            <td style="background-color:#BDB76B;">⬜</td>
61.            <td style="background-color:#BDB76B;">🏳</td>
62.            <td style="background-color:#BDB76B;">💣</td>
63.            <td style="background-color:#BDB76B;">❌</td>
64.         </tr>
65.         <tr>
66.            <td style="background-color:#BDB76B;">❌</td>
67.            <td style="background-color:#BDB76B;">💣</td>
68.            <td style="background-color:#BDB76B;">🏳</td>
69.            <td style="background-color:#BDB76B;">⬅</td>
70.            <td style="background-color:#BDB76B;">⬜</td>
71.            <td style="background-color:#BDB76B;">🏳</td>
72.            <td style="background-color:#BDB76B;">💣</td>
73.            <td style="background-color:#BDB76B;">❌</td>
74.         </tr>
75.         <tr>
76.            <td style="background-color:#BDB76B;">❌</td>
77.            <td style="background-color:#BDB76B;">💣</td>
78.            <td style="background-color:#BDB76B;">🏳</td>
79.            <td style="background-color:#BDB76B;">⬜</td>
80.            <td style="background-color:#BDB76B;">⬜</td>
81.            <td style="background-color:#BDB76B;">🏳</td>
82.            <td style="background-color:#BDB76B;">💣</td>
83.            <td style="background-color:#BDB76B;">❌</td>
84.         </tr>
85.         <tr>
86.            <td style="background-color:#BDB76B;">❌</td>
87.            <td style="background-color:#BDB76B;">💣</td>
88.            <td style="background-color:#BDB76B;">🏳</td>
89.            <td style="background-color:#BDB76B;">⬜</td>
90.            <td style="background-color:#BDB76B;">⬜</td>
91.            <td style="background-color:#BDB76B;">🏳</td>
92.            <td style="background-color:#BDB76B;">💣</td>
93.            <td style="background-color:#BDB76B;">❌</td>
94.         </tr>
95.         <tr>
96.            <td style="background-color:#BDB76B;">❌</td>
97.            <td style="background-color:#BDB76B;">💣</td>
98.            <td style="background-color:#BDB76B;">🏳</td>
99.            <td style="background-color:#BDB76B;">⬜</td>
100.           <td style="background-color:#BDB76B;">⬜</td>
```

```html
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">✕</td>
          </tr>
          <tr>
            <td style="background-color:#BDB76B;">✕</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">☐</td>
            <td style="background-color:#BDB76B;">☐</td>
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">✕</td>
          </tr>
          <tr>
            <td style="background-color:#BDB76B;">✕</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">☐</td>
            <td style="background-color:#BDB76B;">☐</td>
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">✕</td>
          </tr>
          <tr>
            <td style="background-color:#BDB76B;">✕</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">❒</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">✕</td>
          </tr>
          <tr>
            <td style="background-color:#BDB76B;">✕</td>
            <td style="background-color:#BDB76B;">🏁</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">💣</td>
            <td style="background-color:#BDB76B;">🏁</td>
            <td style="background-color:#BDB76B;">✕</td>
          </tr>
          <tr>
             <td style="background-color:#BDB76B;">✕</td>
             <td style="background-color:#BDB76B;">✕</td>
             <td style="background-color:#BDB76B;">✕</td>
             <td style="background-color:#BDB76B;">✕</td>
             <td style="background-color:#BDB76B;">✕</td>
             <td style="background-color:#BDB76B;">✕</td>
             <td style="background-color:#BDB76B;">✕</td>
             <td style="background-color:#BDB76B;">✕</td>
              </tr>
            </table>
            <br>
         </body>
      </html>
```

Y visualmente el programa mostrará lo siguiente:

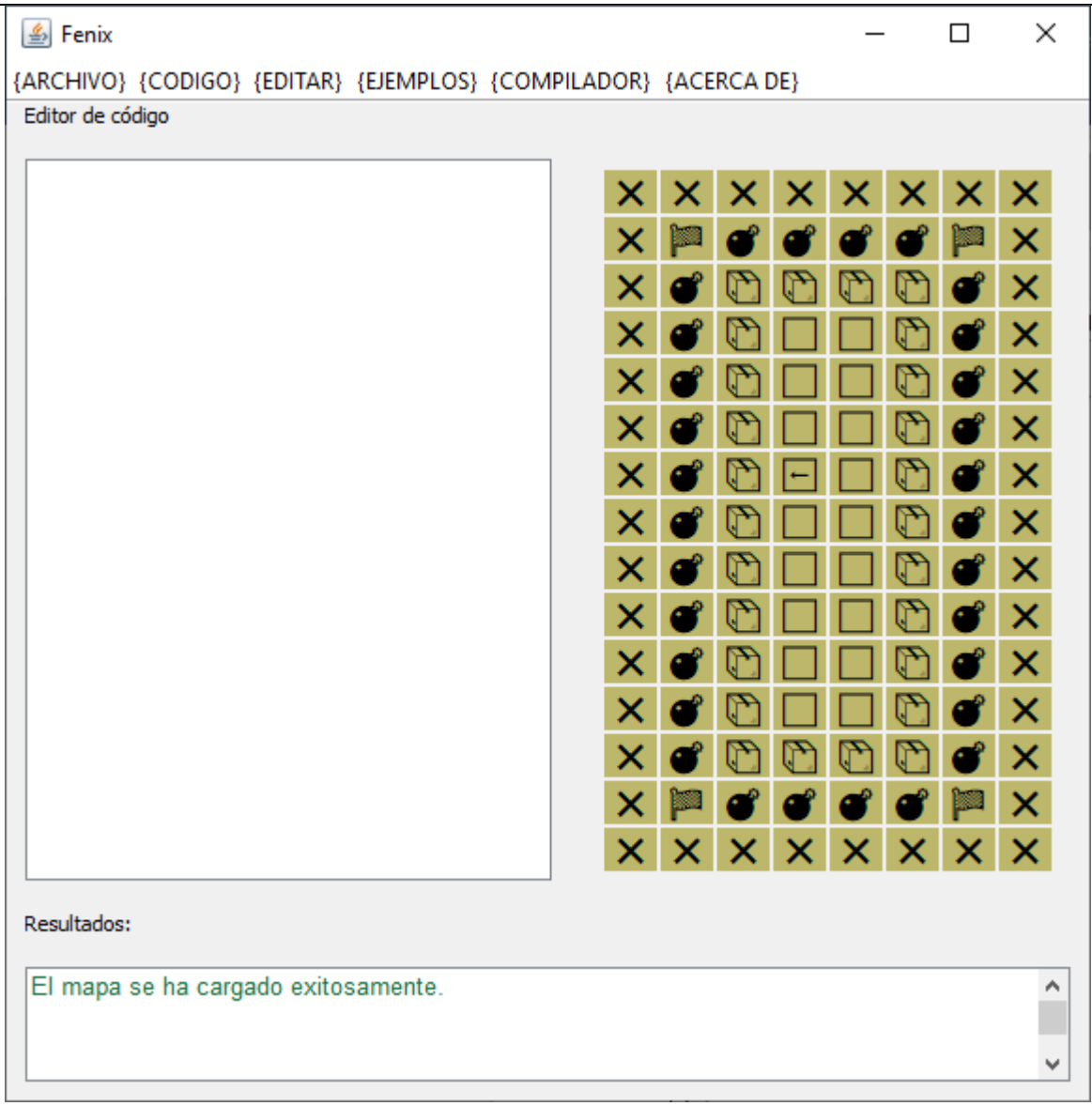

Figura 41 Carga de un Mapa

Con esto se ha finalizado esta parte, el código realizado se reutilizará más adelante.

12 Tabla de símbolos

Antes de ver cómo se construye la máquina virtual es necesario crear un componente llamado Tabla de símbolos.

Esta tabla de símbolos ayudará a obtener valores de las variables en caso de que el usuario las declare y utilice en el programa, además de que contendrá las funciones que se han declarado.

Usualmente la tabla de símbolos es una estructura de datos, pero para este caso se continuará trabajando con JSON debido a que es menor el trabajo a realizar. Se requerirá de un *ArrayList* de objetos *JSONObject*.

```
1.  private final ArrayList<JSONObject> tablaDeSimbolos =new ArrayList<>();
```

La estructura JSON de un elemento de la tabla de símbolos es el siguiente:

```
1.  {
2.    "identifier": "«CADENA DE TEXTO»",
3.    "value": «NUMERO ENTERO»,
4.    "function": "«CADENA DE TEXTO»",
5.    "type": «NUMERO ENTERO»
6.  }
```

Dónde:

- identifier: es el nombre de la variable o función declarada.
- value: contiene el valor de una variable, o el número de parámetros que recibe una función declarada.
- function: es una llave de control, esta se utiliza principalmente para variables temporales de algunas instrucciones de las cuales hablaré más adelante.
- type: es un identificador de control. Si el valor es 0 se trata de una variable, si el valor es 1 se trata de una función declarada.

Ahora los métodos que más se utilizarán son los siguientes:

12.1 Agregar Elemento a la Tabla de Símbolos

Es un método que agrega un elemento a la tabla de símbolos. Para crearlo se requieren de los 4 valores que se han descrito anteriormente.

```
1.      private void agregarElemento(String identificador, int valor,String funcion, int tipo){
2.          tablaDeSimbolos.add(
3.                  new JSONObject()
4.                          .put("identifier",identificador)
5.                          .put("value",valor)
6.                          .put("function",funcion)
7.                          .put("type",tipo)
8.          );
9.      }
```

En este caso agregar un elemento no provocará ningún error, pero se tendrá que estar al pendiente de la tabla de símbolos en caso de que no se pueda crear este elemento.

12.2 Obtener el valor de un elemento de la Tabla de Símbolos

Si se requiere conocer el valor de una variable, en cualquier momento se puede consultar. Tan solo se requiere el nombre del identificador y el tipo.

```
1.  private int valorDeUnElemento(String identificador, int tipo){
2.          int i;
3.          for(i=0; i<tablaDeSimbolos.size();i++){
4.              if(
5.                      (Objects.equals(tablaDeSimbolos.get(i).getString("identifier"),
    identificador)) &&
6.                              (tablaDeSimbolos.get(i).getInt("type") == tipo)
7.              ){
8.                  return (tablaDeSimbolos.get(i).getInt("value"));
9.              }
10.         }
11.         return -999999999;
12.     }
```

Si existe un valor de retorno regresará el valor, de lo contrario regresará el valor “-999999999”, esto se deberá tomar en cuenta para el diseño de la máquina virtual.

12.3 Modificar un valor de un elemento de la Tabla de Símbolos

Se requiere para modificar los valores de la tabla de símbolos. Para lograr una correcta modificación se requiere el nombre del identificador, el valor a modificar y el tipo.

```
1.      private boolean modificarValor(String identificador, int valor, int tipo){
2.          if(!buscarElementoExistente(identificador,tipo)){
3.              return false;
4.          }
5.          for (JSONObject tablaDeSimbolo : tablaDeSimbolos) {
6.              if ((Objects.equals(tablaDeSimbolo.getString("identifier"), identificador))
7.                      && (tablaDeSimbolo.getInt("type") == tipo)) {
8.                  tablaDeSimbolo.put("value", valor);
9.                  return true;
10.             }
11.         }
12.         return false;
13.     }
```

Si la operación se realiza de forma exitosa, el método regresará un booleano positivo, si no realiza la operación regresará un valor negativo.

Estos valores de retorno se tendrán que tomar en cuenta al momento de diseñar la máquina virtual.

12.4 Eliminar elementos de la Tabla de Símbolos

En algunas ocasiones se tendrán que eliminar elementos temporales de la tabla de símbolos, por lo que solo se requiere de un *String* que eliminará todos los elementos que coincidan con la llave “*function*”.

```
1.      private void eliminarElementos(String idVariablesTemporales) {
2.          for(int i=0; i<tablaDeSimbolos.size();i++){
3.              if(Objects.equals(tablaDeSimbolos.get(i).getString("function"),
    idVariablesTemporales)){
4.                  tablaDeSimbolos.remove(i);
5.                  i=0;
6.              }
7.          }
8.      }
```

Tratar de eliminar un elemento no provocará ningún error, al igual que Agregar Elemento se tendrá que estar al pendiente del resultado de la tabla de símbolos cuando se diseñe la máquina virtual.

12.5 Buscar un elemento de la Tabla de Símbolos

En algunas ocasiones se tendrá que buscar un elemento para evitar duplicados y controlar los mensajes de error al usuario. Para esto solo se requiere del nombre del identificador y si se trata de una variable o función.

```
1.      private boolean buscarElementoExistente(String identificador, int tipo){
2.          for (JSONObject tablaDeSimbolo : tablaDeSimbolos) {
3.              if (Objects.equals(tablaDeSimbolo.getString("identifier"), identificador)
4.                      && tablaDeSimbolo.getInt("type") == tipo) {
5.                  return true;
6.              }
7.          }
8.          return false;
9.      }
```

Si encuentra el valor regresa un booleano positivo, de lo contrario regresará un negativo. Se tendrá que tomar en cuenta cuando se diseñe la máquina virtual.

12.6 Limpiar la tabla de símbolos

La limpieza de la tabla de símbolos se realiza usualmente cuando se requiere volver a ejecutar un programa, por lo tanto, se tienen que eliminar todos los elementos sin distinción. Esto se logra con un simple comando que borra todos los elementos del objeto *ArrayList*.

```
1.      public void limpiarTablaDeSimbolos(){
2.          tablaDeSimbolos.clear();
3.      }
```

Estos son los principales métodos que ocupará la tabla de símbolos, con esto se podrá continuar con el diseño de la máquina virtual.

13 Máquina Virtual

Ahora ha llegado el turno del último componente importante del sistema: La máquina virtual. Esta parte del sistema será la encargada de leer el código JSON generado por el compilador, y ejecutará las instrucciones según se indiquen.

13.1 Clase RunCode

Esta clase es la máquina virtual, todos los elementos que se generan cuando se pasa el compilador y el mapa recaen en esta clase. Esta clase es el corazón del proyecto pues es la encargada de resolver el programa que el usuario escribió y mostrará los resultados de una forma gráfica. Es la última parte de este gran proyecto.

13.1.1 Clases necesarias para ejecutar la máquina virtual

La máquina virtual reúne todos los objetos generados con anterioridad, y la creación de nuevos métodos para que estos componentes trabajen en conjunto, para conocer éstos dirigirse al **Capítulo 1.1** del **Apéndice 4: Desarrollo de la máquina virtual.**

13.1.2 Resolver el programa del usuario

El paso siguiente es resolver las instrucciones una por una hasta que no existan más instrucciones por ejecutar o el usuario haya ejecutado una función para terminar el programa. Para esto se tiene el siguiente algoritmo que se encarga de analizar cada una de las instrucciones generadas hasta terminar el programa.

```
1.      private boolean resolverPrograma(String funcionActual, JSONArray subprogram) {
2.          JSONObject temporal;
3.          for (int i = 0; i < subprogram.length(); i++) {
4.              temporal = subprogram.getJSONObject(i);
5.              switch (temporal.getString("instruction")) {
6.                  case "«Palabra Clave»":
7.                      if (!«Llamada al método según sea el caso»()) {
8.                          return false;
9.                      }
10.                     break;
11.                 default:
12.                     return false;
13.             }
14.         }
15.         return true;
16.     }
```

Todas las funcionalidades regresan un valor positivo o negativo. Positivo si la instrucción se ejecutó adecuadamente y negativo si existieron errores en la ejecución.

Cuando regresa un valor negativo el análisis se debe detener e indicar al usuario dónde ocurrió un error y por qué.

Cada una de las instrucciones que se han propuesto solucionar para este proyecto se describen en el **Capítulo 1.2** del **Apéndice 4: Desarrollo de una máquina virtual.**

Con la instrucción final se ha terminado el desarrollo de la máquina virtual.

13.2 Ejecutar la máquina virtual

Ahora se requiere de un hilo que sea capaz de ejecutar la máquina virtual. Debido a que se utilizan en varias ocasiones la instrucción "*Thread.sleep*" es necesario un hilo principal que contenga estas pausas durante la ejecución de la máquina virtual.

La explicación de cómo de estos elementos se explican en el **Capítulo 1.3** del **Apéndice 4: Desarrollo de una máquina virtual.**

Con esto se ha terminado de diseñar el algoritmo para ejecutar la máquina virtual.

14 Detalles finales

Para terminar con el proyecto, es necesario realizar unos ajustes visuales a el programa para mostrar un proyecto terminado.

Estos últimos cambios visuales se encuentran explicados en el **Capítulo 8** del **Apéndice 1: Desarrollo del Entorno de Desarrollo Integrado.**

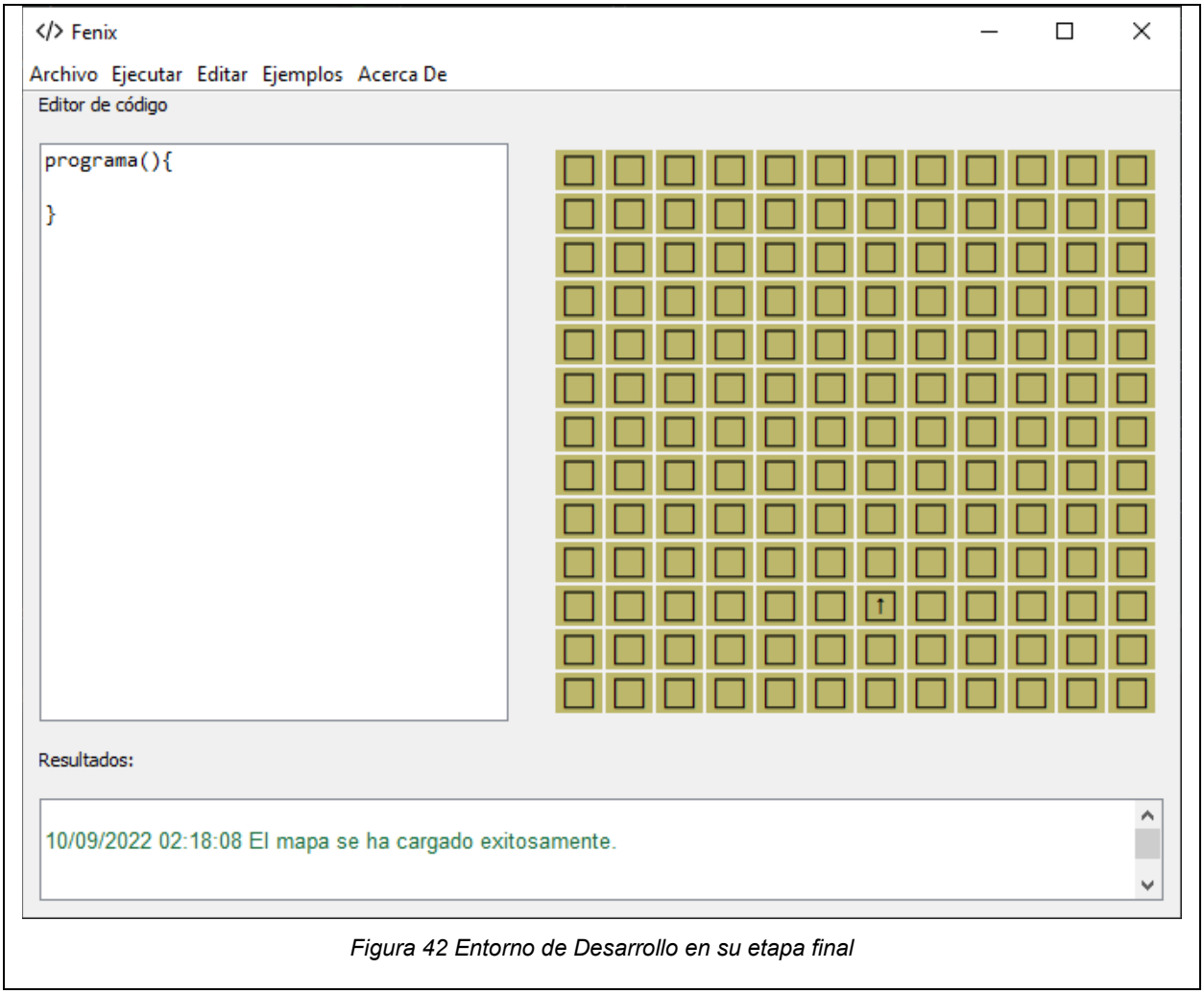

Figura 42 Entorno de Desarrollo en su etapa final

Con estos detalles finales, se finaliza el desarrollo de este proyecto.

A partir de este momento, todo el software entra en una etapa de mantenimiento.

15 Lecciones Aprendidas

Concluyendo este trabajo escrito, en cuanto a los objetivos planteados en la Introducción se puede concluir que se han cumplido los siguientes criterios:

- Se ha desarrollado un compilador propio y único.
- El lenguaje de programación diseñado podría ayudar a personas que se encuentran en los niveles media superior y superior.
- Se desarrolló una máquina virtual para el código generado por el compilador
- Lo anterior, se integró en un Entorno de Desarrollo que muestra al usuario resultados inmediatos.

La herramienta integral se ha completado y funciona en conjunto con el resto de componentes.

Se propuso diseñar un lenguaje de programación, el cual resultó muy fácil de construir pues, enormemente se basó en el lenguaje C. Es decir, se basó en su estructura para desarrollar un nuevo lenguaje, tal y como sucede con la mayoría de lenguajes de alto nivel.

De la parte del compilador, lo único que costó trabajo fue entender cuál iba a ser el lenguaje objeto. Pues no hay guías para crear un compilador, es decir, solo existe documentación de las herramientas más no una guía como tal para solucionar un problema en específico. Afortunadamente, encontré la solución y utilizar JSON como código destino funcionó de forma perfecta, aún sigo sin imaginarme otra solución. Y aunque se lee fácil, encontrar esta solución encontrar esta solución tomó tiempo encontrar.

Sobre el tema de haber desarrollado un compilador para cada idioma creo que podría existir una solución mejor que se implementara desde alguna de las fases de compilación y generación de código, sin embargo, cuando se me ocurrió la idea, el proyecto estaba terminado, por lo que podría ser una oportunidad de mejora.

Lo que me gustó del Entorno de Desarrollo fue esa posibilidad de cambiar de idioma, creo que fue una característica interesante y visualmente potente.

La máquina virtual igualmente fue un componente interesante, pues éste resuelve el programa objeto del compilador. Aunque se comporta como si fuera un intérprete, no lo es. Su funcionamiento resultó muy parecido a lo que realiza el lenguaje Java. Ya que dicho lenguaje realiza un código intermedio antes de ejecutarlo en su máquina virtual. Los retos de realizar una máquina virtual es cómo representar los movimientos en un tablero, cómo moverse a través de él, cómo guardar elementos mientras se va avanzando y luego colocarlos de nuevo sin afectar el tablero en general.

Aunque el verdadero reto del desarrollo de una máquina virtual es resolver el programa principal, afortunadamente dividir cada una de las instrucciones en partes pequeñas ayuda muchísimo a su resolución. Hay que recordar que existen instrucciones que se

tendrán que comunicar con frecuencia en la tabla de símbolos, que implica siempre la consulta y modificación de ésta, y para ciertos casos crear un respaldo de ella o crear una tabla de símbolos temporal. También, siempre estar verificando que los movimientos que realiza el personaje son movimientos válidos y que si se encuentra con el elemento bomba debe terminar la ejecución.

A lo largo del proyecto intenté aplicar conocimiento de Programación Orientada a Objetos (desarrollando el proyecto en base a este paradigma), espero haberlo hecho lo mejor posible. Definitivamente en el desarrollo de esta Tesis tuve un gran crecimiento en comparación a cuando estaba en clases.

Quiero terminar con el trabajo futuro de este proyecto, hay muchas cosas que mejorar como su interfaz o incluso su implementación. Creo que al saber que la máquina virtual termina generando código HTML fue cuando pensé que la mejor opción tuvo que haber sido un sistema web, en lugar de uno de escritorio.

Por último, lo único que falta por probar es la implementación en un salón de clases. Y comprobar que esta la solución propuesta es óptima. Es seguro que existirán errores, y se requiere aprender de ellos para mejor esta solución. Pero hasta que sea implementado se podrá conocer la solución.

16 Índice de Figuras

17 Índice de Enlaces

18 Bibliografía

A. Tucker, & R. Noonan. (2003). *Lenguajes de programación principios y paradigmas .* Madrid: Mc.Graw Hill.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2008). *Compiladores. Principios, técnicas y herramientas. Segunda edición.* México: PEARSON EDUCACIÓN.

Antunez, P. (11 de Abril de 2018). *Analizador Sintactico Ascendente*. Obtenido de Academia: https://www.academia.edu/37523751/Analizador_Sintactico_Ascendente

APKPure. (24 de Octubre de 2019). Sin Título. Obtenido de https://apkpure.com/es/lego%C2%AE-mindstorms-education-ev3/com.lego.education.ev3

Curiel Marquez, R., & Larios Valencia, M. (2005). *C Elementos escenciales.* México: Pearson Educación.

Feng Li , X. (2017). *Advanced Design and Implementation of Virtual Machines.* Taylor & Francis Group, LLC.

HOPCROFT, J. E., MOTWANI, R., & ULLMAN, J. D. (2008). *Introducción a la teoría de autómatas, lenguajes y computación.* Madrid: PEARSON EDUCACIÓN S.A.

Hudson, S. E. (s.f.). *Java(tm) CUP User's Manual*. Obtenido de http://web.mit.edu/javadev/packages/java_cup/

Klein, G., Rowe, S., & Décamps, R. (1998). JFlex User's Manual. Obtenido de https://jflex.de/manual.html

massimomusante. (22 de Diciembre de 2012). What about Alice? (Alice 3.1). Obtenido de https://massimomusante.blogspot.com/2012/12/what-about-alice-alice-31.html

Matte, C. (07 de Mayo de 2018). Kodu Game Lab Review – Game On or Game Over? Obtenido de https://blog.connectedcamps.com/kodu-game-lab-review/

Menchaca García, F. (2002). *Fundamentos de programación en lenguaje C.* México: Fondo de Cultura Económica.

Oracle. (11 de Julio de 2022). *Oracle Cloud Infrastructure*. Obtenido de https://www.oracle.com/mx/database/what-is-json/