LA PRÁCTICA DE LOS LENGUAJES Y AUTÓMATAS EN EL DISEÑO DE LOS COMPILADORES[157]

THE PRACTICE OF LANGUAGES AND AUTOMATS IN THE DESIGN OF COMPILERS

Martha Martínez Moreno[158]

Ofelia Gutiérrez Giraldi[159]

Patricia Horta Rosado[160]

Gabriela Clavel Martínez[161]

Bany Sabel Hernández Cardona[162]

Pares evaluadores: Red de Investigación en Educación, Empresa y Sociedad – REDIEES.[163]

[157] Derivado del proyecto de investigación: Jlefo 1.0 y herramientas de construcción

[158] Licenciada en informática, Tecnológico Nacional de México Instituto Tecnológico de Toluca, profesora de tiempo completo del área de Ingeniería en sistemas computacionales y Tecnologías de información, correo electrónico: martha.mm@toluca.tecnm.mx

[159]Maestra en Ciencias de la Educación y Maestra en Sistemas de Información, Tecnológico Nacional de México, Instituto Tecnológico de Veracruz, profesora investigadora de tiempo completo del área de Ingeniería en Sistemas Computacionales, correo electrónico ofelia.gg@veracruz.tecnm.mx

[160] Maestra en Sistemas de Información, Tecnológico Nacional de México, Instituto Tecnológico de Veracruz, profesora de tiempo completo del área de Ingeniería en Sistemas Computacionales, correo electrónico patricia.hr@veracruz.tecnm.mx

[161] Maestra en Sistemas de Información, Tecnológico Nacional de México, Instituto Tecnológico de Veracruz, profesora de tiempo completo adscrita al área de ciencias básicas, correo electrónico gabriela.cm@bdelrio.tecnm.mx

[162] Maestra en Ciencias Computacionales, Tecnológico Nacional de México, Instituto Tecnológico de Toluca, profesora de tiempo completo del área de Ingeniería en Sistemas Computacionales, correo electrónico bhernandezc@toluca.tecnm.mx

[163] Red de Investigación en Educación, Empresa y Sociedad – REDIEES. www.rediees.org

15.LA PRÁCTICA DE LOS LENGUAJES Y AUTÓMATAS EN EL DISEÑO DE LOS COMPILADORES[164]

Martha Martínez Moreno[165], Ofelia Gutiérrez Giraldi[166], Patricia Horta Rosado[167], Gabriela Clavel Martínez[168], Bany Sabel Hernández Cardona[169]

RESUMEN

Los lenguajes y autómatas son parte de la formación de los futuros ingenieros en sistemas computacionales, con un enfoque de desarrollo de compiladores según el perfil de egreso y el plan de estudios ISIC-2010-224 del Tecnológico Nacional de México. Durante la formación académica del estudiante se presentan dos materias denominadas Lenguajes y autómatas I y Lenguajes y autómatas II, en las cuales es necesario contar con herramientas que permitan el desarrollo de habilidades para la construcción de ese compilador tomando como base un lenguaje de programación, no dejando de lado el manejo de estructuras de datos que complementan la implementación y aplicación de la lógica matemática. De acuerdo a la experiencia y contemplando el ámbito de desempeño de los estudiantes, así como pertenencia de los profesores en este ámbito de desarrollo, en este documento se presentan algunas herramientas utilizadas para la generación de conocimiento en el desarrollo de este ámbito de software de base. Describiendo las herramientas utilizadas, métodos y ejemplos de las etapas de compilación implementadas en proyectos de desarrollo en clase, así como el uso de la aplicación JLEFO y su aportación al proceso de aprendizaje para la materia de lenguajes y autómatas I.

Es importante resaltar que el conocimiento en este ámbito, da al estudiante el soporte en cuanto a la forma de pensar y resolución de problemas dentro del área de las ciencias

[164] Derivado del proyecto de investigación: Jlefo 1.0 y herramientas de construcción

[165] Licenciada en informática, Tecnológico Nacional de México Instituto Tecnológico de Toluca, profesora de tiempo completo del área de Ingeniería en sistemas computacionales y Tecnologías de información, correo electrónico: martha.mm@toluca.tecnm.mx

[166]Maestra en Ciencias de la Educación y Maestra en Sistemas de Información, Tecnológico Nacional de México, Instituto Tecnológico de Veracruz, profesora investigadora de tiempo completo del área de Ingeniería en Sistemas Computacionales, correo electrónico ofelia.gg@veracruz.tecnm.mx

[167] Maestra en Sistemas de Información, Tecnológico Nacional de México, Instituto Tecnológico de Veracruz, profesora de tiempo completo del área de Ingeniería en Sistemas Computacionales, correo electrónico patricia.hr@veracruz.tecnm.mx

[168] Maestra en Sistemas de Información, Tecnológico Nacional de México, Instituto Tecnológico de Veracruz, profesora de tiempo completo adscrita al área de ciencias básicas, correo electrónico gabriela.cm@bdelrio.tecnm.mx

[169] Maestra en Ciencias Computacionales, Tecnológico Nacional de México, Instituto Tecnológico de Toluca, profesora de tiempo completo del área de Ingeniería en Sistemas Computacionales, correo electrónico bhernandezc@toluca.tecnm.mx

computacionales, aporte que es privilegio de la Teoría de la Computación; siendo ésta una materia que cambia el enfoque con que se ven las soluciones.

ABSTRACT:

Languages and automata are part of the training of future computer systems engineers, with a compiler development approach according to the graduation profile and the ISIC-2010-224 study plan of the National Technological Institute of Mexico. During the student's academic training, two subjects called Languages and automata I and Languages and automata II are presented, in which it is necessary to have tools that allow the development of skills for the construction of that compiler based on a programming language, not leaving aside the handling of data structures that complement the implementation and application of mathematical logic. According to the experience and contemplating the field of performance of the students, as well as the belonging of the professors in this field of development, this document presents some tools used to generate knowledge in the development of this field of basic software. Describing the tools used, methods and examples of the compilation stages implemented in development projects in class, as well as the use of the JLEFO application and its contribution to the learning process for the subject of languages and automata I.

It is important to highlight that knowledge in this area gives the student support in terms of thinking and problem solving within the area of computer science, a contribution that is the privilege of Computing Theory; This being a matter that changes the approach with which solutions are seen.

PALABRAS CLAVE: Herramientas, diseño de compiladores, lenguajes, autómatas, implementación, JLEFO. Teoría de la Computación.

Keywords: Tools, compiler design, languages, automata, implementation, JLEFO, Theory of Computation.

INTRODUCCIÓN

Desde los años 90´s en los programas de estudios de la carrera de Ingeniería en sistemas computacionales se contempló el concepto de desarrollo de software de base creando compiladores, de acuerdo a las actualizaciones de las retículas o planes de estudios, se denominaba entonces compiladores, posteriormente programación de sistemas y por último se denominaron lenguajes y autómatas I y II. En diversos programas se fueron utilizando herramientas para desarrollar habilidades y competencias del estudiante en éste ámbito de desarrollo, desde el uso de lenguajes simples, código P, ensambladores, permitiendo crear traductores de lenguajes, basados en muchos de los casos en lenguaje C.
Para ello es necesario conocer las etapas de compilación e ir desarrollando cada una de ellas durante los periodos escolares que abarcan estas materias concluyendo con el diseño e implementación de un proyecto de software de base. En algunos años se consideró la creación de Querys relacionales de bases de datos, pero posteriormente con las actualizaciones del programa de estudios se reconsideró la creación de un compilador basado en un lenguaje de programación.
Luego del uso de herramientas basadas en lenguaje C, como fueron LEX y YACC hasta los principios del año 2000, así como turbo Assambler para el enlace de y generación de códigos intermedios y código objeto o ejecutable, se identificaron y utilizaron nuevas herramientas como JAVA, Java CC, Java Assambler y algunas otras para la construcción de cada etapa de compilación hasta la generación de código ejecutable, las cuales se describirán más adelante.

La teoría de la computación provee conceptos y principios que nos ayudan a entender en general la naturaleza de la disciplina. El campo de las ciencias computacionales incluye un extenso campo de tópicos especiales, desde el diseño de máquinas hasta el ámbito de programación. [Hopcroft, J. E.; Motwani, R.; Ullman, J. D., (2007)] El uso de las computadoras en el mundo real está envuelto por una riqueza de específicos detalles que pueden ser aprendidos a través de estructuras abstractas o modelos computacionales complejos. Estos modelos incorporan las características importantes que son comunes para ambas partes, es decir, hardware y software. Por otro lado, la construcción de modelos es una de las esencias de cualquier disciplina científica y la utilidad de una disciplina a menudo depende de la existencia de teorías y leyes simples pero poderosas.

Además, las aplicaciones que se pueden obtener con el uso de estas estructuras corresponden al campo del diseño digital, lenguajes de programación y compiladores, entre muchas otras.

Pero para aprender a dominar estas estructuras de las que se ha hablado, necesitamos conocer sus fundamentos básicos y después incursionar a la comprensión de las mismas.

Lenguajes

Las lenguas son sistemas más o menos complejos, que asocian contenidos de pensamiento y significación a manifestaciones simbólicas tanto orales como escritas. Aunque en sentido estricto, el lenguaje sería la capacidad humana para comunicarse mediante lenguas, se suele usar para denotar los mecanismos de comunicación no humanos (el lenguaje de las abejas o el de los delfines), o los creados por los hombres con fines específicos (los lenguajes de programación, los lenguajes de la lógica, los lenguajes de la aritmética). (Lenguajes naturales y lenguajes formales Francisco José Salguero, 1995).

Se puede definir el lenguaje como un conjunto de palabras. Cada lenguaje está compuesto por secuencias de símbolos tomados de alguna colección finita. En el caso de cualquier lengua natural (castellano, inglés, francés...), la colección finita es el conjunto de las letras del alfabeto junto con los símbolos que se usan para construir palabras (tales como el guion, el apóstrofe en el caso del inglés...). De forma similar, la representación de enteros, son secuencias de caracteres del conjunto de los dígitos {0,1,2,3,4,5,6,7,8,9}.

La única restricción importante sobre lo que puede ser un lenguaje es que todos los alfabetos son finitos. De este modo, los lenguajes, aunque pueden tener un número infinito de cadenas, están restringidos a que dichas cadenas estén formadas por los símbolos que definen un alfabeto finito y prefijado.

Gramáticas

Para estudiar los idiomas matemáticamente, necesitamos un mecanismo para describirlos. El lenguaje cotidiano es impreciso y ambiguo, por lo que las descripciones informales en inglés o español a menudo son inadecuadas. A medida que avanzamos, aprenderemos sobre varios mecanismos de definición de lenguaje que son útiles en diferentes circunstancias. Por ello tenemos un concepto común y poderoso, la noción de una gramática,

por ejemplo, la Ilustración 2. Una gramática para el idioma español nos dice si una oración en particular está bien formada o no. [Hopcroft, J., Motwani, R., & Ullman, J. (2007)]

Una gramática G se define como una cuádrupla:

G = (V, T, S, P)

Donde:

- ✓ V es un conjunto finito de objetos llamados variables.
- ✓ T es un conjunto finito de objetos llamados símbolos terminales.
- ✓ S ∈ V es un símbolo especial llamado la variable de inicio.
- ✓ P es un conjunto finito de producciones.

Se supondrá sin más mención que los conjuntos V y T son no vacíos y disjuntos. Las reglas de producción son el corazón de una gramática; especifican cómo la gramática transforma una cadena en otra y, a través de ella, definen un lenguaje asociado con la gramática. En nuestra discusión asumiremos que todas las reglas de producción son de la forma x _y

Donde x es un elemento de (V ∪ T) + y y está en (V ∪ T) *. Las producciones se aplican de la siguiente manera: Dada una cadena w de la forma W = uxv.

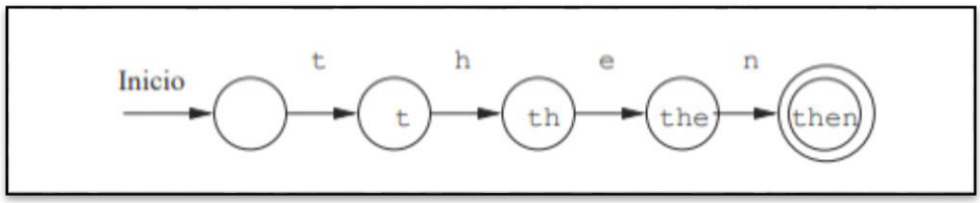
Autómata

Un autómata es un modelo abstracto de una computadora digital. Como tal, cada autómata incluye algunas características esenciales como podemos ver en la figura 1. Tiene un mecanismo para leer la entrada. Se supondrá que la entrada es una cadena sobre un alfabeto dado, escrita en un archivo de entrada, que el autómata puede leer, pero no cambiar. El archivo de entrada está dividido en celdas, cada una de las cuales puede contener un símbolo. El mecanismo de entrada puede leer el archivo de entrada de izquierda a derecha, un símbolo a la vez. El mecanismo de entrada también puede detectar el final de la cadena de entrada (al detectar una condición de fin de archivo).

El autómata puede producir resultados de alguna forma. Puede tener un dispositivo de almacenamiento temporal, que consiste en un número ilimitado de celdas, cada una capaz de contener un solo símbolo de un alfabeto (no necesariamente el mismo que el alfabeto de

entrada). El autómata puede leer y cambiar el contenido de las celdas de almacenamiento. Finalmente, el autómata tiene una unidad de control, que puede estar en cualquiera de un número finito de estados internos, y que puede cambiar de estado de alguna manera definida. [Hopcroft, J., Motwani, R., & Ullman, J. (2007)]

Figura 1. Modelo de autómata para el reconocimiento de la palabra then.



Lenguajes Regulares Y Expresiones Regulares

- Lenguajes Regulares

  Los lenguajes regulares, de tipo 3 según la jerarquía de Chomsky, son aquellos que son reconocidos por autómatas de estados finitos, son denotados por expresiones regulares y generados por gramáticas regulares. Estos lenguajes contienen a todos los lenguajes finitos generados a partir de cualquier alfabeto. Los lenguajes finitos tipificados como regulares poseen ciertas propiedades que lo caracterizan y distinguen de otros lenguajes más complejos. [Hopcroft, J., Motwani, R., & Ullman, J. (2007)]

- Expresiones Regulares

  Una expresión regular describe un conjunto de cadenas sin enumerar sus elementos. Toda expresión regular básicamente se puede ver como una cadena formada por las letras del alfabeto y ciertos operadores como lo vemos en la Ilustración 8. Para que la cadena sea considerada una expresión regular deberá seguir una construcción gramatical correcta.

Lenguaje de programación JAVA

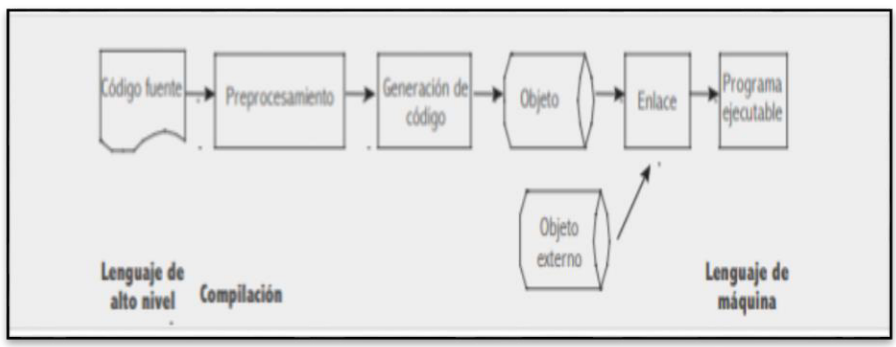
Java es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995 por Sun Microsystems. Hay muchas aplicaciones y sitios web que no funcionarán a menos que tenga Java instalado y cada día se crean más. Java es rápido,

seguro y fiable. Desde portátiles hasta centros de datos, desde consolas para juegos hasta súper computadoras, desde teléfonos móviles hasta Internet, Java está en todas partes. Como cualquier lenguaje de programación, el lenguaje Java tiene su propia estructura, reglas de sintaxis y paradigma de programación. El paradigma de programación del lenguaje Java se basa en el concepto de programación orientada a objetos (OOP), que las funciones del lenguaje soportan. El lenguaje Java es un derivado del lenguaje C, por lo que sus reglas de sintaxis se parecen mucho a C: por ejemplo, los bloques de códigos se modularizan en métodos y se delimitan con llaves ({y}) y las variables se declaran antes de que se usen.

Estructuralmente, el lenguaje Java comienza con paquetes. Un paquete es el mecanismo de espacio de nombres del lenguaje Java. Dentro de los paquetes se encuentran las clases y dentro de las clases se encuentran métodos, variables, constantes, entre otros. [Java, t. (2018).]

En el ámbito de las computadoras, los algoritmos se expresan mediante lenguajes de programación, como C, Pascal, Fortran o Java (entre muchos otros). Sin embargo, esta representación no es suficiente, ya que el microprocesador necesita una expresión mucho más detallada del algoritmo, que especifique en forma explícita todas las señales eléctricas que involucra cada operación. La tarea de traducción de un programa desde un lenguaje de programación de alto nivel hasta el lenguaje de máquina se denomina compilación, y la herramienta encargada de ello es el compilador. A continuación, en la figura 2 se pueden distinguir las etapas más importantes: [Gustavo López Ismael Jeder, Augusto Vega].

Figura 2. Etapas de compilación. Gustavo López.



Se observan en las figuras 3 y 4 la diversidad de diagramas que ilustran los procesos de compilación desde el enfoque de cada autor, considerando todos los autores un proceso de análisis y de síntesis o resultados, tomando como base un programa fuente y como resultado o salida un programa objeto o ejecutable.

Figura 3. Etapas de compilación.

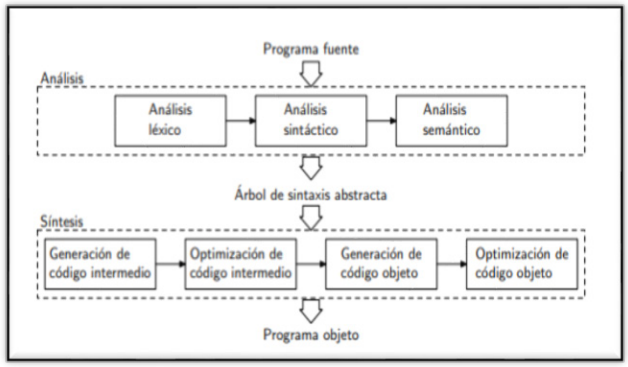
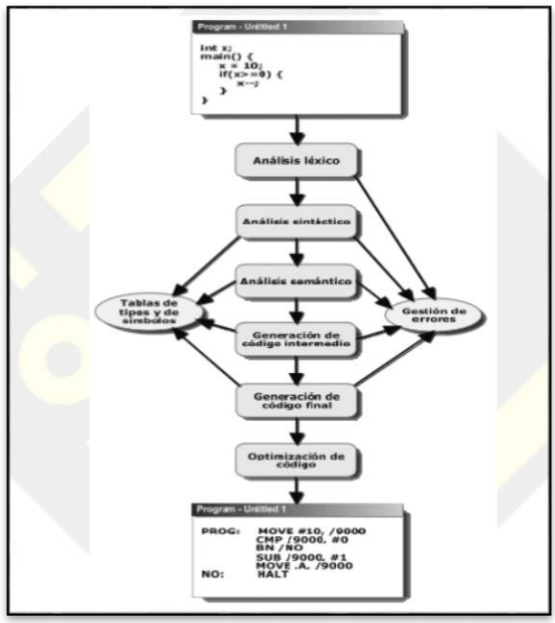


Figura 4. Etapas de compilación Ruiz Catalán.



JAVA CC

En tiempos pasados, las herramientas para la implementación de las estructuras de datos en la resolución de problemas de lógica matemática, específicamente durante las etapas

de análisis de un lenguaje de programación, era Lex y Yacc; estas herramientas se usaban para desarrollar fase léxica (Lex) y sintáctica (Yacc) respectivamente. Sin embargo, a principios del presente siglo se migró al uso de Java CC (Java Compiler Compiler) considerando como plataforma a Java.

Java CC es un generador de analizadores sintácticos de código abierto para el lenguaje de programación Java, licenciado bajo BSD (Berkeley Software Distribution). Genera un parser para una gramática presentada en notación BNF (Bakus Naur Form) y se diferencia de Yacc en que su salida es en código Java. Otra diferencia radica en que genera analizadores descendentes (Top-Down) de tipo LL(K) implicando que la recursión por la izquierda no se pueda realizar. El constructor de árboles que utiliza es JJTree, el cual basa su construcción de árboles de abajo hacia arriba.

En la actualidad, la gran ventaja de que los estudiantes utilicen Java CC como herramienta se ve reflejada en el hecho de que manejan Java, por lo que la comprensión del código que se genera en las fases léxica y sintáctica les resulta bastante más sencilla de comprender. Aunado a ello, enlazar Java CC con código Java para el desarrollo de entornos gráficos también permite facilidad en la programación.

DESARROLLO

Durante la formación profesional y académica del estudiante de ingeniería en sistemas computacionales, según el programa de estudios del Tecnológico Nacional de México, se contempla el desarrollo de un proyecto integrador de las materias lenguajes y autómatas I y II, el cual se inicia con la etapa de análisis del proceso de compilación y concluye en la etapa de síntesis en la segunda materia.

Según el proceso de compilación y las referencias mencionadas en la introducción, es importante que el estudiante y el profesor además de identificar las bases teóricas de cada etapa, se vayan implementando cada una de ellas en el proyecto de desarrollo.

Por lo que el desarrollo de este trabajo describirá cada etapa y las herramientas utilizadas de apoyo para la construcción del compilador.

I. JLEFO y Análisis léxico.
II. Análisis sintáctico.
III. Análisis semántico.

IV. Generación de código Intermedio.

V. Optimización.

VI. Generación de código ejecutable u objeto.

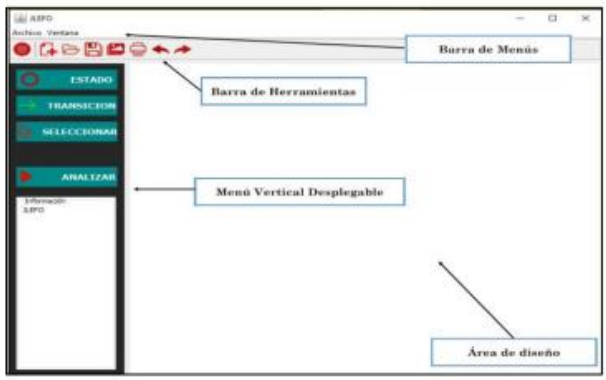
JLEFO

Durante el desarrollo del proyecto, en la materia denominada lenguajes y autómatas I se desarrollan competencias para la creación de expresiones regulares a partir de un autómata y para este caso, se desarrolló la aplicación JLEFO versión 1.0, es un software de escritorio diseñado durante el 2019 como herramienta de apoyo para la materia de lenguajes y autómatas I, la cual permite la creación de Expresiones regulares basados en un autómata finito determinístico y resuelve la expresión regular una vez analizada la cadena o cadenas de entrada. Esta versión está a disposición del profesor y del alumno para mejor desempeño y comprensión de este tema en el proceso de aprendizaje, la cual ha permitido disminuir los índices de reprobación enfocados en los temas específicos de expresiones regulares y gramáticas. Tema que sin duda es esencial para el diseño del compilador.

La interfaz principal como se puede observar en la figura 5, muestra la vista principal de la herramienta JLEFO que cuenta con los siguientes componentes:

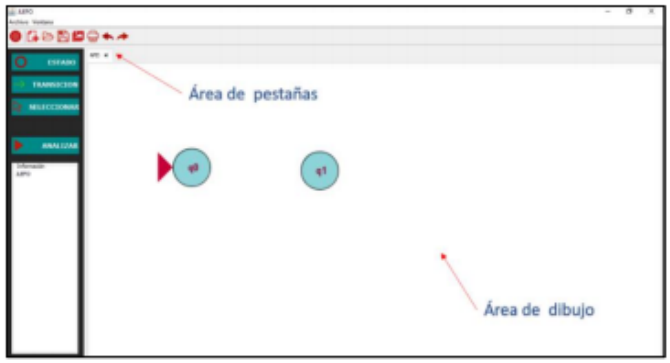
Menú vertical desplegable: contiene las opciones que permiten interactuar con los módulos AFD, AFND, ER. Barra de herramientas: contiene opciones para interactuar con los diferentes módulos. Barra de menú: permite elegir el módulo con el cual trabajar. Nota: Por default no se tiene un módulo establecido para que se ejecute al abrir la aplicación, por lo que el usuario tendrá que elegirlo.

Figura 5. Interfaz de JLEFO.



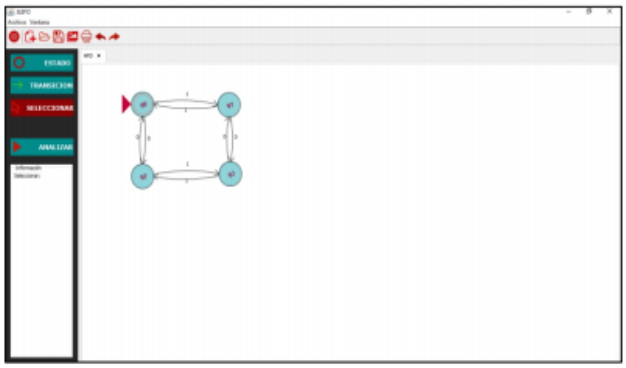
La interfaz de Dibujo permite que el usuario elija un módulo inmediatamente se genera una pestaña junto con un área de dibujo que permite diagramar un AFD o AFND. Para construir el autómata, se utiliza el menú vertical desplegable y el mouse, el cual permite insertar todos los elementos que contienen este tipo de estructuras como lo muestra la figura 6.

Figura 6. Área de dibujado.



En la figura 7 se observa el diagrama de un ejemplo de un ejercicio de un autómata finito determinista que se puede analizar con la herramienta. El cual corresponde a siguiente lenguaje: Lenguaje= {Cadenas sobre el alfabeto 0 y 1 que contengan el mismo número par de 1´s y 0´s.}

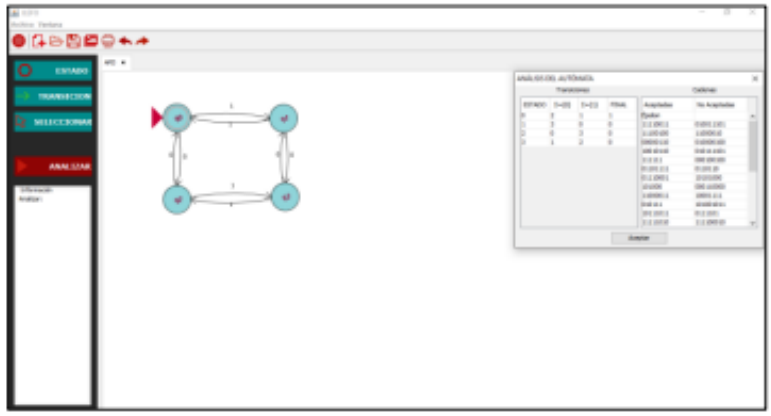
Figura 7. Autómata finito determinístico diseñado con JLEFO



Interfaz de Análisis de Cadenas

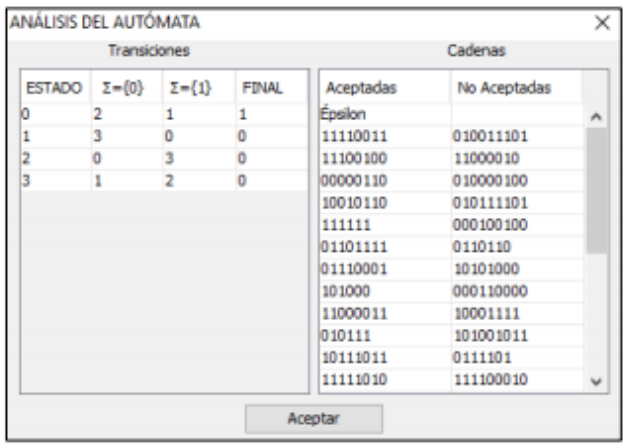
El análisis de cadenas es la característica principal de la herramienta, para realizar el análisis el usuario debe seleccionar la opción de analizar que se encuentra dentro del menú vertical, y en automático la aplicación le mostrara mediante una interfaz que se puede observar en la figura 8 el resultado del análisis ejecutado.

Figura 8. Resultado del análisis de cadenas.



Los elementos principales de la interfaz de análisis de cadenas son los siguientes y que además se muestran en la figura 9. Tablas de transición: Esta tabla muestra como el autómata paso de un estado a otro, los elementos que componen esta tabla son los mismos que un autómata de manera gráfica, es decir, estado, transición y elementos del alfabeto. Tabla de cadenas: Esta tabla contiene cadenas aceptadas y rechazas de diferentes longitudes de caracteres, además de que se considera si el autómata acepta la cadena de épsilon.

Figura 9. Análisis de cadenas.



Análisis léxico.

En esta etapa de inicio del proceso de compilación, se debe considerar el momento de apertura o lectura de un archivo o código fuente, ver figura 8, para lo cual se pueden

utilizar toda instrucción de control de archivos, en el caso de los proyectos desarrollados se ha utilizado Java como primer herramienta y aunque JavaCC es parte de la misma tecnología, el manejo de archivos y la lectura del mismo, se realiza utilizando clases de entrada y salida de datos, ya que es el momento de crear paralelamente la denominada tabla de tokens o parser identificados ver figura 10. Esto significa que una vez que los archivos se logran identificar, abrir y leer su contenido, deberá iniciar el proceso de análisis de esas cadenas, símbolos o elementos previamente definidos en el lenguaje a través de expresiones regulares.

Para crear las expresiones regulares, se utiliza la herramienta JavaCC con el formato especificado donde se describen los símbolos, cadenas, palabras reservadas e incluso cadenas no válidas para el lenguaje, creando un archivo con extensión.jj. El formato de las expresiones regulares es simple, pues define los caracteres o símbolos permitidos al crear la cadena aceptada. Ver figura 10.

Una vez que se definen los parsers o expresiones regulares en esta herramienta, se obtienen las cadenas de entrada y se comparan con cada expresión regular creada.

En algunos casos, al crearlo de forma manual las cadenas o símbolos del lenguaje (si se usa puramente Java) se tendrán que identificar cada cadena o secuencia de símbolos aceptados por el lenguaje a través de un número para posteriormente comparar la secuencia de cadenas y crear la tabla de sintaxis. Ver figura 6 para el manejo de Archivos desde su lectura y creación de resultados.

Figura 10. Creación de tokens en Javacc.

```
options
{
    STATIC = false;
}

PARSER_BEGIN(Comp)

public class Comp {

}

PARSER_END(Comp)

SKIP :
{
    " " |
    "\t" |
    "\n" |
    "\r"
}

TOKEN :
{
    <MAIN: "main" >{Lexico.lexico.append("MAIN ->"+ image + "\n");}
}
TOKEN :
{
    <CLASS: "class" > {Lexico.lexico.append("CLASS ->"+ image+ "\n");}
|
    <ABSTRACT: "abstract" > {Lexico.lexico.append("ABSTRACT ->"+ image+ "\n");}
|
    <EXTENDS: "extends" > {Lexico.lexico.append("EXTENDS ->"+ image+ "\n");}
|
    <IMPLEMENTS: "implements" > {Lexico.lexico.append("IMPLEMENTS ->"+ image+ "\n");}
|
    <IMPORT: "import" > {Lexico.lexico.append("IMPORT ->"+ image+ "\n");}
|
    <STATIC: "static" > {Lexico.lexico.append("STATIC ->"+ image+ "\n");}
}
TOKEN :
{
    <FOR: "for" > {Lexico.lexico.append("FOR ->"+ image+ "\n");}
|
    <IF: "if" > {Lexico.lexico.append("IF ->"+ image+ "\n");}
|
    <ELSE: "else" > {Lexico.lexico.append("ELSE ->"+ image+ "\n");}
|
    <WHILE: "while" > {Lexico.lexico.append("WHILE ->"+ image+ "\n");}
|
    <DO: "do" > {Lexico.lexico.append("DO ->"+ image+ "\n");}
|
```

En la figura 11 se crea la tabla resultante del análisis léxico con Java, bien sea que se utilice Javacc como herramienta o puramente Java.

Figura 11. Manejo de archivos para etapa léxica en Java.

```
public class ManejoDeArchivos {

    public static void guardarArchivo(String texto) {
        FileWriter fichero = null;
        PrintWriter pw = null;
        try {
            fichero = new FileWriter("files\\archivo.txt");
            pw = new PrintWriter(fichero);
            pw.println(texto);
        } catch (IOException ex) {
            Logger.getLogger(ManejoDeArchivos.class.getName()).log(Level.SEVERE, null, ex);
            System.err.println("Error al guardar archivo por defecto."
                    + "\tClase: ManejoDeArchivos"
                    + "\tMétodo: guardarArchivo");
        } finally {
            if (fichero != null) {
                try {
                    pw.close();
                    fichero.close();
                } catch (IOException ex) {
                    Logger.getLogger(ManejoDeArchivos.class.getName()).log(Level.SEVERE, null, ex);
                    System.err.println("Error al guardar archivo por defecto."
                            + "\tClase: ManejoDeArchivos"
                            + "\tMétodo: guardarArchivo");
                }
            }
        }
    }
```

Figura 12. Clase léxico y su resultado usando Java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package compilador;

import java.util.Hashtable;


public class Lexico
{
    public static  StringBuilder lexico = new StringBuilder();

    public static String salida()
    {
        return lexico.toString();
    }
}
```

Finalmente, al diseñar el proceso de análisis léxico, se utilizan conocimientos y habilidades desarrolladas en la estructuración, lectura y creación de archivos, por ejemplo, en la figura 12 se observa una clase para cargar un archivo fuente.

Figura 12. Cargar archivo.

```
public static String cargarArchivo() {
    String cadena = "";
    File archivo = null;
    FileReader fr = null;
    BufferedReader br = null;

    archivo = new File("files\\archivo.txt");
    try {
        fr = new FileReader(archivo);
        br = new BufferedReader(fr);
        String linea;
        while ((linea = br.readLine()) != null) {
            cadena += linea + "\n";
        }
    } catch (FileNotFoundException ex) {
        Logger.getLogger(ManejoDeArchivos.class.getName()).log(Level.SEVERE, null, ex);
        System.err.println("Error al cargar archivo por defecto."
                + "\tClase: ManejoDeArchivos"
                + "\tMétodo: cargarArchivo");
    } catch (IOException ex) {
        Logger.getLogger(ManejoDeArchivos.class.getName()).log(Level.SEVERE, null, ex);
        System.err.println("Error al cargar archivo por defecto."
                + "\tClase: ManejoDeArchivos"
                + "\tMétodo: cargarArchivo");
    } finally {
        if (fr != null) {
            try {
                br.close();
                fr.close();
            } catch (IOException ex) {
                Logger.getLogger(ManejoDeArchivos.class.getName()).log(Level.SEVERE, null, ex);
                System.err.println("Error al cargar archivo por defecto."
                        + "\tClase: ManejoDeArchivos"
                        + "\tMétodo: cargarArchivo");
            }
        }
    }
```

En la mayoría de los casos en que le proyecto en su etapa léxica se diseña la tabla de transiciones para comprobar la lectura de cadenas de entrada para el siguiente paso que es la etapa sintáctica, se crean clases para identificar a cada cadena o expresión regular con un número de identificación y así comprobar la secuencia al formar las gramáticas. Ver figura 13.

Figura 13. Numeración de tokens.

```
public class Lexico {

    String lexema;
    String nombre;
    int numero;

    //Diccionario de tokens
    String dic[][] =
    {
        {"int","Tipo de Dato","1"},
        {"double","Tipo de Dato","1"},
        {"String","Tipo de Dato","1"},
        {"char","Tipo de Dato","1"},
        {"println","Palabra reservada println","2"},
        {"whl","Palabra reservada whl","3"},
        {"if","Palabra reservada if","4"},
        {"case","Palabra reservada case","5"},
        {"readln","Palabra reservada readln","6"},
        {"shift","Palabra reservada shift","7"},
        {"default","Palabra reservada default","8"},
        {"break","Palabra reservada break","9"},
        {"<","Operador relacional","10"},
        {">","Operador relacional","10"},
        {"<=","Operador relacional","10"},
        {">=","Operador relacional","10"},
        {"==","Operador relacional","10"},
        {"!=","Operador relacional","10"},
        {"+","Operador aritmetico","11"},
        {"-","Operador aritmetico","11"},
        {"*","Operador aritmetico","11"},
        {"/","Operador aritmetico","11"},
        {"^","Operador aritmetico","11"},
        {"&&","Operador logico","12"},
        {"||","Operador logico","12"},
        {"|","Pleca","13"},
        {"!","Exclamacion","14"},
        {"#","Numero / Gato","15"},
        {"(","Parentesis abre","16"},
```

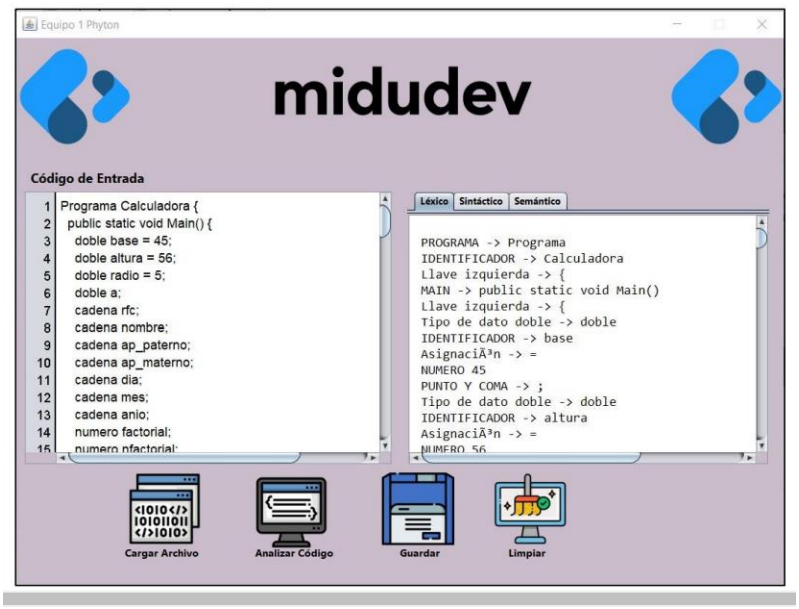
Para poder crear la secuencia del autómata que representa las cadenas posibles del lenguaje., Ver figura 14.

Figura 14. Creación de la matriz del autómata.

```
public Lexico etiquetar(String palabra) {
    Lexico objLex = new Lexico();
    objLex.lexema = palabra;
    int i = 0;
    boolean ban = false;
    while (i < objLex.dic.length)
    {
        if (palabra.equals(dic[i][0]))
        {
            ban = true;
            objLex.nombre = dic[i][1];
            objLex.numero = Integer.parseInt(dic[i][2]);
            break;
        }
        i++;
    }
    if (ban)
    {
        return objLex;
    }

    //Automata General
    int matTran [][] =
    {
        {1,-1,2,-1,5,8,10},
        {1,1,1,-1,-1,-1,-1},
        {-1,-1,2,3,-1,-1,-1},
        {-1,-1,4,-1,-1,-1,-1},
        {-1,-1,4,-1,-1,-1,-1},
        {6,6,6,-1,-1,-1,-1},
        {-1,-1,-1,-1,7,-1,-1},
        {-1,-1,-1,-1,-1,-1,-1},
        {8,8,8,-1,-1,9,-1},
        {-1,-1,-1,-1,-1,-1,-1},
        {10,10,10,-1,-1,-1,11},
        {-1,-1,-1,-1,-1,-1,-1}

    };
```

El resultado de la validación léxica en el diseño se muestra en la figura 15. con un código fuente (declaración de variables), del lado derecho podemos observar el resultado del analizador léxico, donde se etiqueta según corresponde el nombre de cada símbolo o declaración.

Figura 15. Comprobación léxica.



Análisis sintáctico.

En esta etapa el objetivo es analizar las cadenas de entrada y comparar con la secuencia de instrucciones válidas según las gramáticas diseñadas, para ello se pueden desarrollar nuevamente en Java o utilizando métodos en Javacc con la regla a producir y la opción del mensaje de error utilizando las instrucciones de Java u otras producciones válidas utilizando el operador or. Recordando que por cada gramática producida se debe continuar con la lectura de las expresiones del código fuente hasta encontrar el final de instrucción o de archivo. Ver figura 16. En la mayoría de los lenguajes el final de instrucción está dado por el símbolo punto y coma.

Figura 16. Diseño de gramáticas en Javacc.

```
void Programa() :
{
   TokenAsignaciones.SetTables();
}
{
       <PROGRAMA>{xd.CodigoOptimizado.cadenas.add(new xd.CodigoOptimizado(token.image,token.kind));}<IDENTIFICADOR>{xd.CodigoOptim
       Sentencias()
       <RBRACE>{xd.CodigoOptimizado.cadenas.add(new xd.CodigoOptimizado(token.image,token.kind));}
       <RBRACE>{xd.CodigoOptimizado.cadenas.add(new xd.CodigoOptimizado(token.image,token.kind));}
       <EOF>

}

void Sentencias():
{}
{
       sent_2()
}

void sent_2():
{}
{
       (

              SentenciaDeclaracion()
              |LOOKAHEAD(2)SentenciaDeclaracion2()
              |SentenciaCall()
              |SentenciaIf()
              |SentenciaElse()
              |SentenciaFor()
              |SentenciaWhile()
              |SentenciaDoWhile()
              |SentenciaFuncion()

              |SentenciaRead()
```

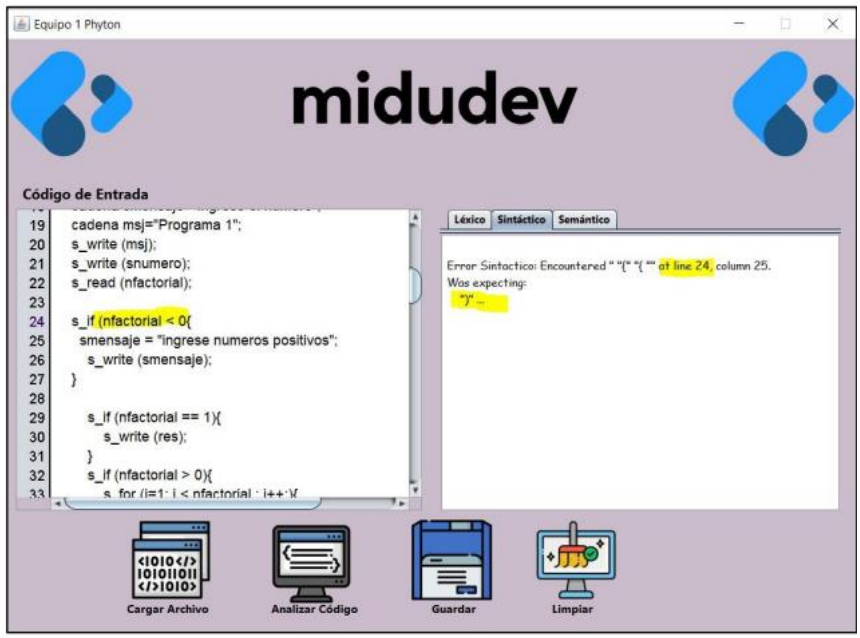
En el análisis sintáctico se realiza la validación de los identificadores, tipos de dato, etc. con ayuda de los tokens y la gramática. En la implementación del Análisis Sintáctico se puede observar que en la figura 17 se encuentran el método que valida los errores Sintáctico, este se encuentra en la clase llamada TokenMgrError.

Figura 17. Método para validación de errores sintácticos.

```
protected static String LexicalError(boolean EOFSeen, int lexState, int errorLine, int errorColumn, String errorAfter, char curChar) {
  return("Lexical error at line " +
        errorLine + ", column " +
        errorColumn + ".  Encountered: " +
        (EOFSeen ? "<EOF> " : ("\"" + addEscapes(String.valueOf(curChar)) + "\"") + " (" + (int)curChar + "), ") +
        "after : \"" + addEscapes(errorAfter) + "\"");
}
public String getMessage() {
  return super.getMessage();
}
public TokenMgrError() {
}
public TokenMgrError(String message, int reason) {
  super(message);
  errorCode = reason;
}
public TokenMgrError(boolean EOFSeen, int lexState, int errorLine, int errorColumn, String errorAfter, char curChar, int reason) {
  this(LexicalError(EOFSeen, lexState, errorLine, errorColumn, errorAfter, curChar), reason);
}
```

El resultado del análisis sintáctico se muestra en la figura 18.

Figura 18. Resultado sintáctico.



Análisis semántico

Esta etapa tiene como objetivo validar los tipos de datos asociados en todo el código fuente, desde su declaración, asignación, operación, condición o comparación, así como la entrada y salida de los datos, de tal forma que el código debe validar si los identificadores, variables, estructuras o incluso métodos están siendo correctamente utilizados según el propósito para el cual fueron creados.

Por ejemplo, si una variable se declaró con un tipo de dato entero, booleano, char o string, que el valor que reciban, o las operaciones que realicen sean precisamente con esos tipos de datos asociados, así mismo el resultado obtenido al asignarse en otro identificador corresponda al declarado.

Los tipos de datos dependen del diseño del lenguaje, la permicidad y la asociación entre ellos a partir de la creación de una tabla en la que se define la compatibilidad entre ellos y el resultado que se obtendrá.

Algunos lenguajes de programación como Python no requieren una previa declaración de tipos de datos y al operar el identificador se conoce su tipo asociado, sin embargo, si el identificador cambia de valor o tipo nuevamente al operar, no indica ningún error y toma el último valor asociado a la variable o identificador.

También es necesario resaltar que en el análisis semántico se incorpora el tema de jerarquía de operación, al momento de resolver una expresión aritmética se iniciará con la operación de mayor a menor jerarquía y esto debe implementarse a través de rutinas o métodos que validen la jerarquía de operación.

Por lo que la etapa semántica debe implementar estas dos validaciones en un código fuente, la validación por herencia y asociación de tipos de datos y la jerarquía de operación. Para el caso práctico y herramientas utilizadas, se continúa con Java y en esta etapa de análisis semántico se validarán la declaración de variables, duplicidad entradas y salidas. [Márquez, L., 2013]

Implementación del Análisis Semántico.

En la figura 19 podemos observar el código del método que lleva por nombre “chekAsing”. En el cual se validan las máscaras o imágenes de los tokens, validando así cada uno de ellos, entrando en el ciclo if correspondiente, si la imagen del token es correcta no

regresa ningún valor, en caso de ser incorrecto se manda un mensaje al usuario con el error, la línea en la que se encuentra el error y que se debe hacer para poder corregir dicho error.

Figura 19. Método semántico.

En la figura 20 se puede observar la continuación de las líneas de código utilizadas en el método “chekAsing”, en las cuales se sigue haciendo la validación de los tokens usando las imágenes que con anterioridad se le colocó a cada uno de los tokens, continuando con la evaluación de cada uno de ellos, se puede observar la validación de ciclos, entradas, salidas y asignaciones principalmente, siendo ese el objetivo principal del análisis semántico.

Figura 20. Validación de tipo de dato.

```
    if (intComp.contains(tipoIdent2)) {
        return " ";
    } else //Si el tipo de dato no es compatible manda el error
    {
        Interface.txtSemantico.append("Error: No se puede convertir " + TokenAsig.image + " a Entero \r\nLinea: " + TokenIzq.beginLine);
        return "Error: No se puede convertir " + TokenAsig.image + " a Entero \r\nLinea: " + TokenIzq.beginLine;
    }
} else if (tipoIdent1 == 2) //double
{
    if (decComp.contains(tipoIdent2)) {
        return " ";
    } else {
        Interface.txtSemantico.append("Error: No se puede convertir " + TokenAsig.image + " a Decimal \r\nLinea: " + TokenIzq.beginLine);
        return "Error: No se puede convertir " + TokenAsig.image + " a Decimal \r\nLinea: " + TokenIzq.beginLine;
    }
} else if (tipoIdent1 == 4) //char
{
    /*variable segunda: cuenta cuantos datos se van a asignar al caracter:
                        si a el caracter se le asigna mas de un dato (ej: 'a' + 'b') marca error
                        NOTA: no se utiliza un booleano ya que entraria en asignaciones pares o impares*/
    segunda++;
    if (segunda < 2) {
        if (chrComp.contains(tipoIdent2)) {
            return " ";
        } else {
            Interface.txtSemantico.append("Error: No se puede convertir " + TokenAsig.image + " a Caracter \r\nLinea: " + TokenIzq.beginLine);
            return "Error: No se puede convertir " + TokenAsig.image + " a Caracter \r\nLinea: " + TokenIzq.beginLine;
        }
    } else //Si se esta asignando mas de un caracter manda el error
    {
        Interface.txtSemantico.append("Error: No se puede asignar mas de un valor a un caracter \r\nLinea: " + TokenIzq.beginLine);
        return "Error: No se puede asignar mas de un valor a un caracter \r\nLinea: " + TokenIzq.beginLine;
    }

} else if (tipoIdent1 == 3) //string
{
    if (strComp.contains(tipoIdent2)) {
        return " ";
    } else {
        Interface.txtSemantico.append("Error: No se puede convertir " + TokenAsig.image + " a Cadena \r\nLinea: " + TokenIzq.beginLine);
        return "Error: No se puede convertir " + TokenAsig.image + " a Cadena \r\nLinea: " + TokenIzq.beginLine;
    }
} else {
    Interface.txtSemantico.append("Ultimo El Identificador " + TokenIzq.image + " no ha sido declarado" + " Linea: " + TokenIzq.beginLine);
    return "Ultimo El Identificador " + TokenIzq.image + " no ha sido declarado" + " Linea: " + TokenIzq.beginLine;
```

En la figura 21 se puede observar el método “chekVariable” el cumple la función de verificar si un identificador ha sido o no declarado, primero intenta obtener el token a verificar (el token debe de estar en la tabla de tokens), si el token pasa por la validación el método regresa una línea en blanco esto se debe a que el token y la validación son correctos, si el método encuentro un error o no lo puede obtener manda un mensaje de error al usuario identificando en que línea a sucedido dicho error y que es lo que el usuario puede hacer para solucionarlo.

Figura 21. variable declarada.

```
/*Metodo que verifica si un identificador ha sido declarado,
        ej cuando se declaran las asignaciones: i++, i--)*/
public static String checkVariable(Token checkTok) {
    try {
        //Intenta obtener el token a verificar(checkTok) de la tabla de los tokens
        int tipoIdent1 = (Integer) tabla.get(checkTok.image);
        Interface.txtSemantico.append("Error: El identificador " + checkTok.image + " ya ha sido declarado \r\nLinea: " + checkTok.beginLine);
        return "Error: El identificador " + checkTok.image + " ya ha sido declarado \r\nLinea: " + checkTok.beginLine;
    } catch (Exception e) {
        //Si no lo puede obtener, manda el error
        return " ";
    }
}
```

En la figura 22 y 23 se muestran errores semánticos, la causa de ello es el tipo de dato, el cual está declarado como entero y variable duplicada.

Figura 22. Validación semántica.

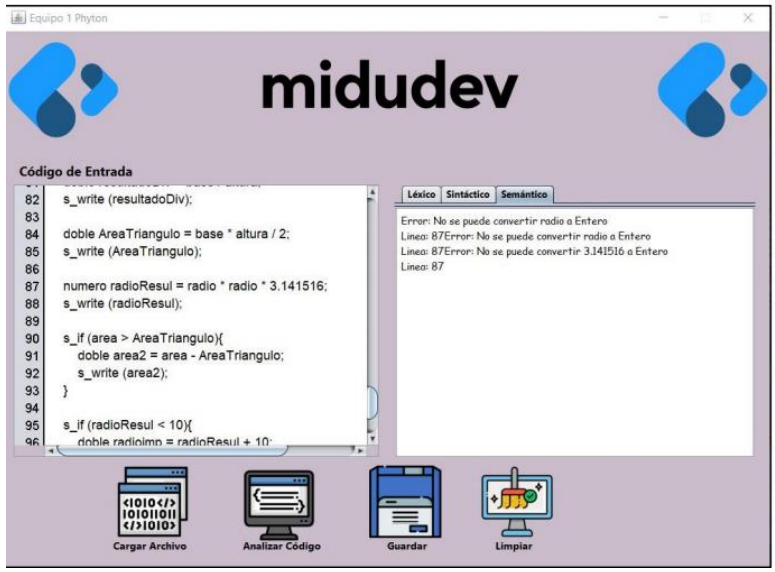
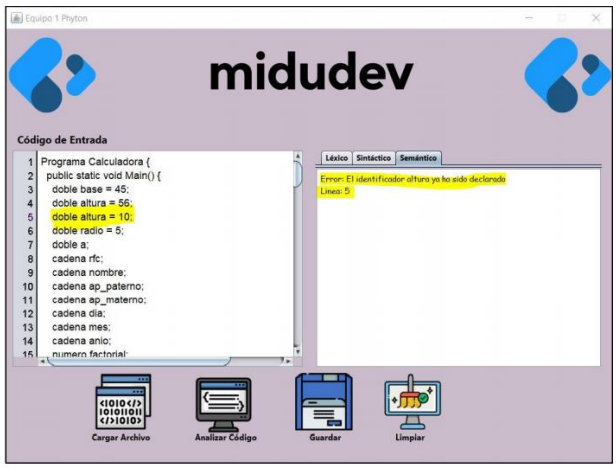


Figura 23 duplicidad en declaración.



Generación de código intermedio

En esta fase de compilación se debe generar una traducción de las instrucciones de alto nivel a código intermedio, puede utilizarse lenguaje ensamblador en su versión de Intel o incluso un ensamblador de Java denominado JavaAssambler, el cual a través de una máquina virtual de Java se crea la traducción a código intermedio y de ahí se genera el ejecutable en ambos casos.

Pero además se pueden traducir expresiones únicamente aritméticas implementando mecanismos a través de lenguaje P, o código en 3 direcciones o cuádruplos, partiendo de notaciones infijas a traducción pre o posfija. [Anónimo.]

Para este caso se utilizó la traducción de expresiones aritméticas infijas a notaciones posfijas.

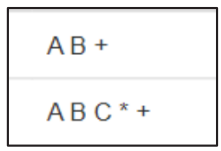
Igual se continúa con Java la implementación de métodos de traducción.

El objetivo de las notaciones es expresar operaciones aritméticas, como se debe existen varias notaciones para representar expresiones matemáticas, que se diferencian en el orden en que se escriben los argumentos (operandos) de los operadores. Las más relevantes son: prefija, infija y postfija.

Posfija

En notación sufija (figura 24), la expresión sería A B C * +. Una vez más, el orden de las operaciones se conserva, ya que el * aparece inmediatamente después de la B y la C, denotando que el * tiene precedencia, con el + apareciendo después. Aunque los operadores se movieron y ahora aparecen antes o después de sus respectivos operandos, el orden de cada operando se mantuvo exactamente igual en relación con los demás. [11]

Figura 24. Notaciones posfijas.



Representaciones de código intermedio

La representación de código intermedio consta de 2 fases, la primera son los cuádruplos la cual consta de separar las operaciones del código fuente y realizarlas por este mismo método, la segunda fase es el postfijo la cual consta de acomodar las operaciones resultantes en este mismo orden imprimiendo un resultado legible para el usuario.

Cuádruplos

Un cuádruple es una estructura tipo registro con cuatro campos que se llaman (op, result, arg1, arg2). El campo op contiene un código interno para el operador. Por ejemplo, la proposición de tres direcciones x = y + z se representa mediante el cuádruplo (ADD, x, y, z).

Las proposiciones con operadores unarios no usan el arg2. Los campos que no se usan se dejan vacíos o tienen un valor NULL. Como se necesitan cuatro campos se le llama representación mediante cuádruplos. [9]

Código o método utilizado para la traducción.

La traducción a código intermedio fue realizada con la ayuda de ciertos métodos por ejemplo el método denominado generaCuadrupla (figura 25), el cual con la ayuda de arrayList y de pilas compara y realiza dicha generación de código, el cual ayuda al control y generación de un código limpio y entendible.

Figura 25. método para código intermedio.

```
for (Lexema lexema : expresionCopy) {

    if (lexema.is(NumeroTokens.NUMERO_ENTERO) || lexema.is(NumeroTokens.NUMERO_REAL) || lexema.is(NumeroTokens.VARIABLE)) {

        operandos.push(lexema);

        // lexema.is(Lexema.OPERADOR)
    } else if (lexema.is(NumeroTokens.OPERADORES_ARITMETICOS)) {
        operadores.push(lexema);

        if (operandos.size() >= 2) {

            operador = operadores.pop();
            operadorDerecha = operandos.pop();
            operadorIzquierda = operandos.pop();

              aux = new Lexema("T" + contadorTemporales, 0, 0, "41");
            aux = new Lexema("T" + contadorTemporales, NumeroTokens.VARIABLE);
            operandos.push(aux);
            cuadruplas.add(new Cuadrupla(operador, operadorIzquierda, operadorDerecha, aux));

            contadorTemporales++;

        }

    } else if (lexema.is(NumeroTokens.OPERADOR_ASIGNACION)) {
        Lexema l1 = operandos.pop();
        Lexema l2 = operandos.pop();

        System.out.println(l2.getLexema() + " " + lexema.getLexema() + " " + l1.getLexema());
        Interface.txtCuadruples.append(l2.getLexema() + " " + lexema.getLexema() + " " + l1.getLexema() + "\n\n");
        contadorTemporales = 1;
    }
}
return cuadruplas;
```

Estructuras utilizadas para código intermedio

Las estructuras utilizadas para el programa fueron en su mayoría las pilas, las cuales ayudaron a la comparación del código y así poder determinar el orden de este. Se puede observar los métodos Comparar y PrintCuadruplas (figura 26), estos métodos cumplen la función de comparar lo que llevan dentro y así facilitar la generación del código intermedio. Los métodos antes mencionados también cumplen la función de ordenar la entrada y salida de los operadores que el código fuente contenga, dando a si salida a un código intermedio en postfijo.

Figura 26. cuádruplas.

```
public static void printCuadruplas(ArrayList<Cuadrupla> cuadruplas) {
    for (Cuadrupla cuadrupla : cuadruplas) {
        System.out.println(cuadrupla);
    }
}

/**
 * Comparar si 2 cuadruplas son iguales operandos y operacion
 *
 * @param c1 cuadrupla 1
 * @param c2 cuadrupla 2
 * @return true si son iguales
 */
public static boolean comparar(Cuadrupla c1, Cuadrupla c2) {

    if (c1.operacion.getLexema().equals(c2.operacion.getLexema())
            && (c1.operacion.getLexema().equals("*") || c1.operacion.getLexema().equals("+")))
        if (c1.operando1.getLexema().equals(c2.getOperando2().getLexema())
                && c1.operando2.getLexema().equals(c2.getOperando1().getLexema())) {
            return true;
        }
    }

    return c1.operacion.getLexema().equals(c2.operacion.getLexema()) && c1.operando1.getLexema(
            .equals(c2.operando1.getLexema()) && c1.operando2.getLexema()
            .equals(c2.operando2.getLexema());
}

public String getEtiqueta() {
    return etiqueta;
}

public void setEtiqueta(String etiqueta) {
    this.etiqueta = etiqueta;
}
```

Jerarquía de operación.

Para realizar dicha jerarquía de operación fue necesario hacer uso de un vector llamado “MAP_JER” (figura 27), el cual contiene todos los valores posibles en la jerarquía, a los valores se les asigna un número para poder ser comparados más adelante, todo dependerá de si el número es mayor o menor que el anterior, así se podrá dar entrada a la jerarquía de postfijo.

Figura 27. jerarquía de operación.

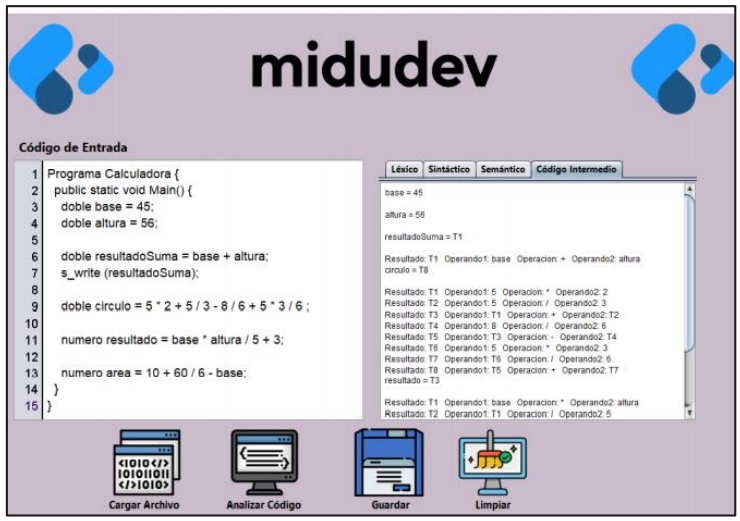
```
private static final HashMap<String, Integer> JERARQUIA = new HashMap<>();
private static final Object[][] MAP_JER = {
    {"||", 1},
    {"&&", 2},
    {"!", 3},
    {"=", 4},
    {"==", 4},
    {">", 4},
    {"<", 4},
    {">=", 4},
    {"<=", 4},
    {"!=", 4},
    {"+", 5},
    {"-", 5},
    {"*", 6},
    {"/", 6},
    {"^", 7},
    {"(", 9},
    {")", 9}
};
```

Resultados Generación de código intermedio

Se puede apreciar un ejemplo de 15 líneas (figura 28), el cual será analizado obteniendo el código intermedio del mismo, el resultado obtenido será mostrado al usuario

en la pestaña que lleva por nombre “código intermedio”, el código obtenido 34 es arrojado en formato de cuádruples, con la generación de variables nuevas las cuales llevan las operaciones realizadas en el código fuente.

Figura 28. Resultado código intermedio.



Optimización de código

Esta etapa está relacionada con el uso de recursos del propio compilador, en la mayoría de los casos no es visible para el usuario, pues el diseñador del compilador decide el código que va a optimizar, si el código intermedio resultante creando expresiones aritméticas más cortas, árboles abstractos para reducir las expresiones y para ello deberá tener más habilidades en el manejo de ensamblador, o bien, crear algún método que permita optimizar el código fuente eliminando todo aquello que el programador escribe y que no se utiliza al momento de la traducción y ejecución. Por ejemplo, los comentarios, variables declaradas y no usadas, los enter o tabs en el código fuente, ya que, a mayor número de líneas, el tamaño del archivo fuente es mayor y así mismo el tiempo de ejecución y uso de recursos en memoria o registros.

Para este caso, se sugiere diseñar métodos o técnicas que eliminen este tipo de líneas de código fuente y mostrar los tamaños de los archivos una vez optimizados.

Después de que el programa ya realiza la generación del código intermedio procedemos a recuperar todo el código fuente mediante sus lexemas, esto lo podemos lograr

agregando una clase llamada “CodigoOptimizado” y agregando unas pocas líneas en el archivo analyzer.jj del proyecto. La clase “CodigoOptimizado” contará con los atributos necesarios para cada lexema. En este caso un lexema, donde se almacenará la imagen de este y el número de token para poder identificar más fácilmente con qué tipo de lexema estamos tratando, ya sean tipos de datos, identificadores, etc.

Recuperación del código fuente

Dentro de la siguiente clase tenemos un “arraylist” (figura 29) del tipo de dato de la misma clase donde vamos almacenando todos los tokens, así creando una lista con todos los tokens. Cabe recalcar que para esto omitimos los espacios en blanco, de forma que nos regrese todos los tokens en una sola línea.

Eliminación de variables sin uso

Para realizar la eliminación de variables sin uso procedemos a recolectar todas las variables del código e ingresarlas en una lista. Para esto vamos a recorrer todo el arreglo de lexemas que obtuvimos con la clase “CodigoOptimizado” y separaremos las variables en un nuevo arreglo, como se muestra en el siguiente ciclo realizado para poder recorrer todo el arreglo (figura 30).

Figura 29. Código optimización.

```
public class CodigoOptimizado {

    public static ArrayList<CodigoOptimizado> cadenas    = new ArrayList();

    public static ArrayList<CodigoOptimizado> getLexemas() {
        return cadenas;
    }

    public static void setLexemas(ArrayList<CodigoOptimizado> aLexemas) {
        cadenas = aLexemas;
    }

    private String lexema;
    private int token;

    public CodigoOptimizado(String lexema, int token) {
        this.lexema = lexema;
        this.token = token;
    }

    public int getToken() {
        return token;
    }

    public void setToken(int token) {
        this.token = token;
    }

    public String getLexema() {
        return lexema;
    }

    public void setLexema(String lexema) {
        this.lexema = lexema;
    }

    public boolean is(int... token) {
        if (token.length == 1) {
            return token[0] == this.token;
        }

        for (int i : token) {
            if (this.token == i) {
                return true;
            }
```

Posterior a esto procedemos a recorrer ese mismo arreglo para encontrar los elementos que no se repiten. En el siguiente método se realizan dos ciclos figura 31, uno que toma las variables y otro que recorre todo el arreglo buscando otra que sea igual, si este último encuentra uno igual aumenta el contador de variable repetida y remueve la variable del arreglo.

Figura 30. eliminación variables.

```
for (CodigoOptimizado lexema : codigoOptimizado) {
    if (lexema.is(NumeroTokens.VARIABLE)) {
        Variables.add(lexema.getLexema());
    }
}
```

Una vez que ya tenemos todas las variables que se repiten dentro del arreglo independiente, ahora, hacemos uso del método “EliminarRepetidas” el cual busca las variables de dicho arreglo en todo el código.

Figura 31. Variables repetidas.

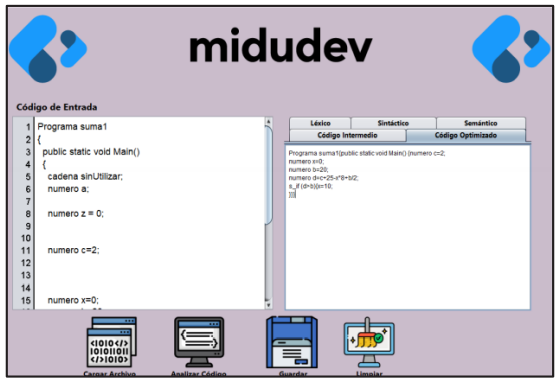
```
System.out.println("\n\n");
for (int i = 0; i < Variables.size(); i++) {
    for (int j = i+1; j < Variables.size(); j++) {
        if (Variables.get(i).equals(Variables.get(j))) {
            contadorVariables++;
            Variables.remove(j);

        }
    }
    if (contadorVariables == 0 && nombre >0) {
        VariablesRepetidas.add(Variables.get(i));
        System.out.println(Variables.get(i) + " Variable no utilizada");
    }
    nombre++;
    contadorVariables = 0;
}
xd.EliminarRepetidas.EliminacionDeRepetidas(VariablesRepetidas,codigoOptimizado);
```

Una vez que ya recibimos todo el código, pero sin espacios recibiremos todo el código en una sola línea y sin espacios. Para ordenar esto tenemos que agregar ciertos espacios. Por ejemplo, antes y después de un identificador y un salto de línea después de cada punto y coma; después de esto mandamos cada lexema a la parte gráfica para que el usuario pueda visualizarlo.

Resultados de la optimización de código

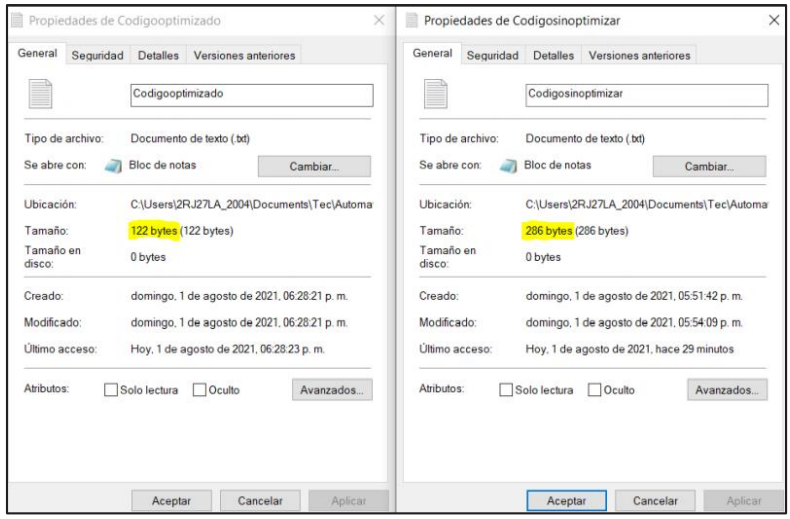
Para probar la optimización de código, se utilizó un código fuente de un tamaño de 40 líneas, al momento de optimizarlo el código bajó a un total de 6 líneas, ya que se eliminaron variables no utilizadas, espacios en blanco y se recorrió el código (figura 32), todo esto para mejorar el rendimiento de memoria y el tiempo de ejecución, además de ser un poco más estético y agradable al momento de programar.

Figura 32. Resultado de optimización.



En cuanto al peso de los archivos también se puede notar una gran diferencia en su peso (figura 33), con este ejemplo podemos entender que la optimización de código no solo sirve para reducir el número de líneas en un código fuente sino también reduce el tamaño de dichos archivos.

Figura 33. Tamaño de archivos.



Código ejecutable

Para la creación del archivo ejecutable partimos de que el compilador ha sido diseñado con herramientas de Java, las cuales generan archivos .jar, y dependerá del usuario final que tenga las herramientas necesarias para su ejecución. Por lo que para crear un ejecutable y el proceso de instalación del software resultante, en este caso el compilador, se

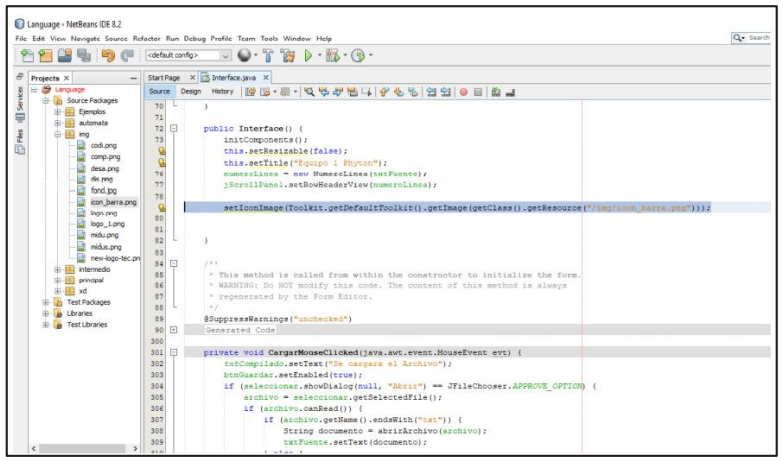
requiere de la conversión del archivo .jar a .exe y luego el vaciado de archivos, librerías o ejecutable a la ruta que especifique el usuario.

Inserción de icono para la aplicación

Para poder utilizar una imagen como icono de la aplicación debemos identificar la clase utiliza en nuestro entorno de desarrollo como principal, en nuestro caso la clase principal tiene por nombre: Interface. Después de haber identificado la clase principal, se debe agregar una imagen en la carpeta donde se contienen las imágenes para el funcionamiento del proyecto. Para después agregar la siguiente línea de código en el constructor de dicha clase.

setIconImage(Toolkit.getDefaultToolkit().getImage(getClass().getResource("/img/icon_barra.png"))); En la parte que se encuentra subrayada se colocará la dirección de nuestra imagen. Ver figura 34.

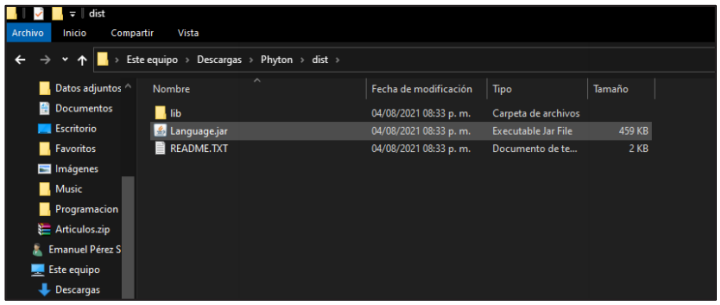
Figura 34. Inserción del ícono.



Creación de archivo .jar

Para crear un archivo con extensión .jar en nuestro entorno de desarrollo debemos ir a la opción clean and build lo cual generará de forma automática una carpeta en nuestro proyecto con el nombre de dist en el cual encontramos el .jar de nuestro proyecto junto con las librerías necesarias para su correcto funcionamiento. Ver figura 35.

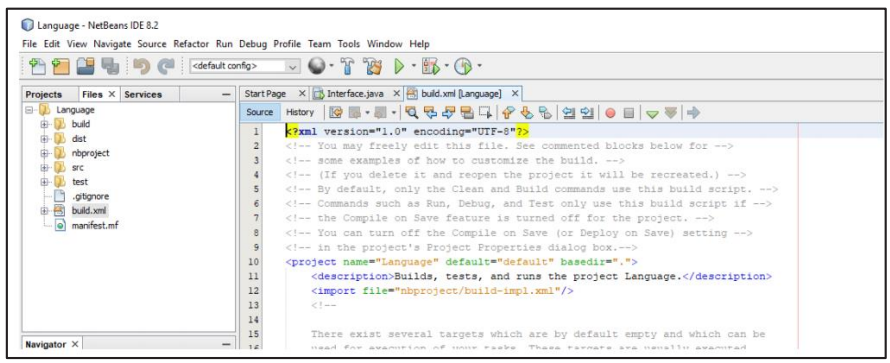
Figura 35. ubicación del .jar.



Comprimir librerías

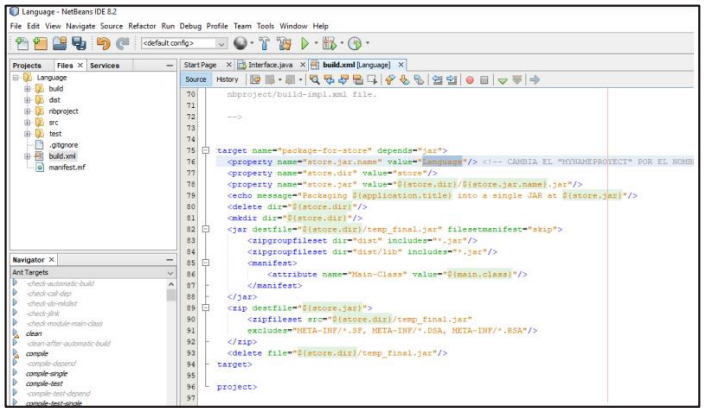
Contando con los archivos que se han generado anteriormente el proyecto puede funcionar perfectamente, el problema con el que aún se cuenta es que para poder instalarlo se deben enviar el proyecto con extensión .jar y el archivo de las librerías, estas dos deben ser introducidas en una carpeta, ya que si el proyecto no cuenta con las librerías no podrá funcionar. El siguiente paso será comprimir las librerías en el archivo jar de nuestro proyecto, para lo cual en el entorno de desarrollo estando en la carpeta de nuestro proyecto seleccionaremos la pestaña files, en las opciones mostradas iremos a build.xml. Figura 36.

Figura 36. Ubicación Build.xml.



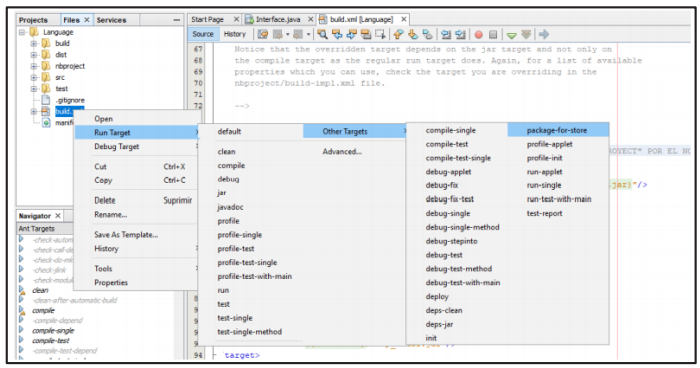
Posteriormente nos iremos a la parte final y antes de cerrar la etiqueta project> colocaremos el código que a continuación se muestra, tendremos que cambiar la parte subrayada por el nombre de nuestro proyecto. Ver figura 37.

Figura 37. modificación del xml.



Después de agregar el código en el archivo build.xml daremos un click derecho y seleccionamos las siguientes opciones: figura 38.

Figura 38. opciones del xml.



Como resultado de estos pasos se obtiene una carpeta que se genera automáticamente en la carpeta de nuestro proyecto con el nombre store en la cual ya contiene las librerías necesarias para su funcionamiento.

Generar ejecutable

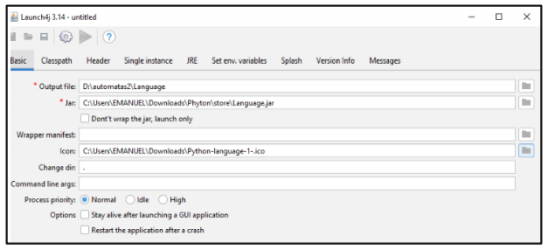
Para generar el ejecutable de nuestro proyecto haremos uso de un software con el nombre Launch4j en el cual colocaremos las siguientes opciones:

• Output file: será la ubicación donde se guardará el ejecutable (el nombre del proyecto debe ser con extensión .exe).

• Jar: cargaremos el archivo jar generado anteriormente en el cual ya se incluyen las librerías.

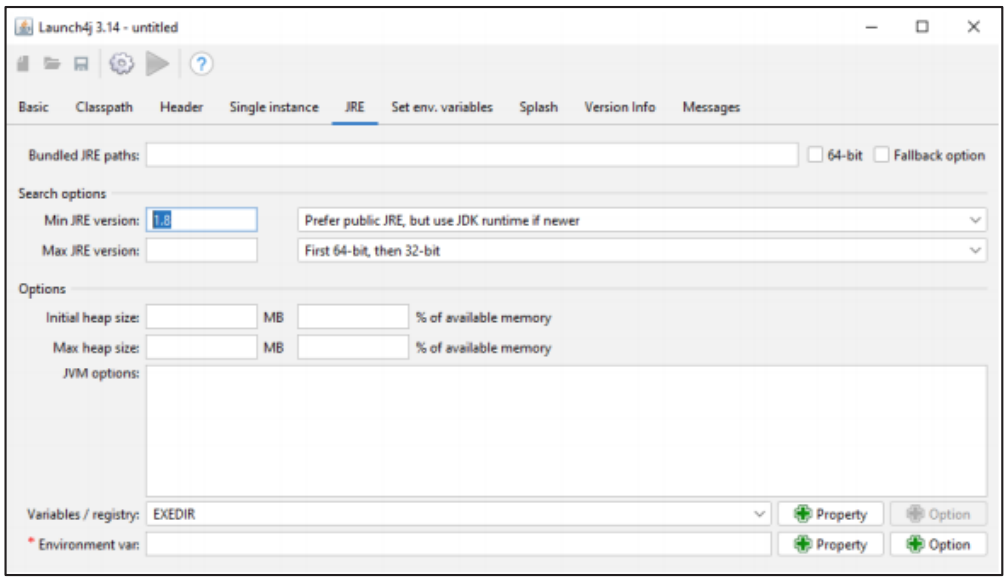
• Icon: Agregaremos una imagen que será utilizada como icono del proyecto el cual debe ser con extensión. Icon Figura 39

Figura 39. Carga de archivos.



Posteriormente nos iremos a la pestaña JRE en la opción Min JRE versión en donde únicamente colocaremos la versión mínima de java que se necesitara para que el proyecto pueda ser ejecutado. Para nuestro caso el mínimo necesario es 1.8. Finalmente le damos click en el engrane de la aplicación para que comience la generación del archivo ejecutable. Figura 40.

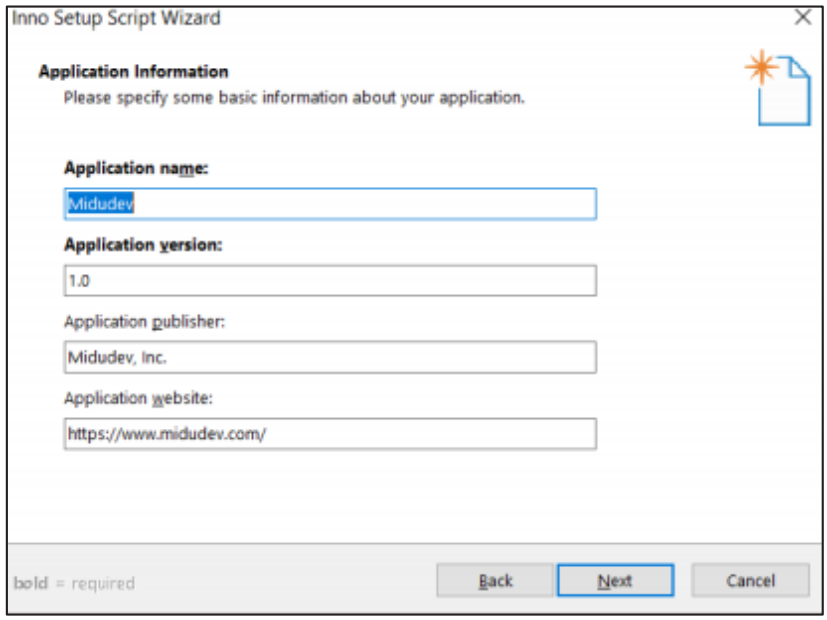
Figura 40. Generación ejecutable con Launch4j.



Generación de archivo setup.

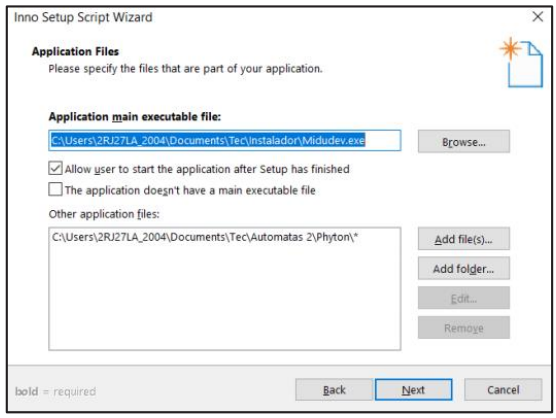
Una vez realizado el ejecutable, procederemos a usar la aplicación de Inno Setup (figura 41), iniciaremos dando un nombre a nuestra aplicación, una versión y un nombre de un sitio web de ejemplo.

Fig. 41 Inno Setup.



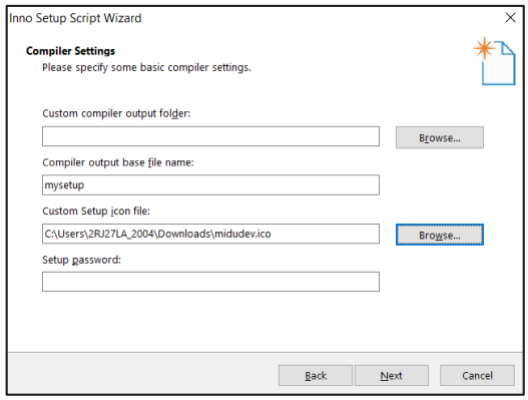
En la siguiente pestaña se deberá seleccionar el ejecutable .exe antes ya creado, incluyendo además todas las carpetas necesarias que utilizará nuestra aplicación (figura 42).

Figura 42. Archivos necesarios.



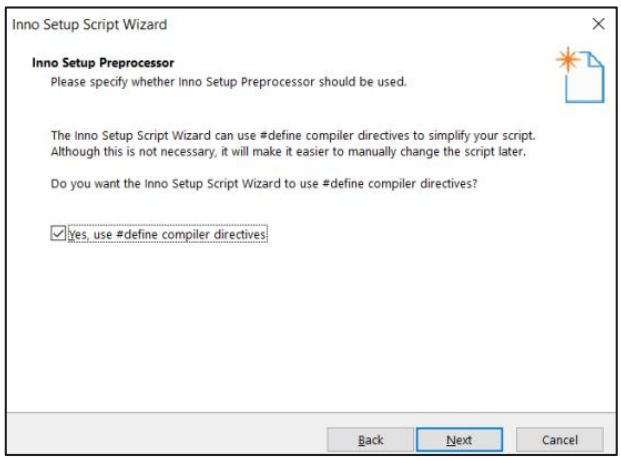
En la siguiente pestaña mostrará en que carpeta se guardará el instalador y el nombre de dicho instalador figura 43.

Figura 43. Carpeta y nombre del instalador.



Por último, seleccionaremos la opción de "finish" y compilaremos el programa. Como paso final nos dirigiremos a la carpeta en la cual guardamos el setup de nuestra aplicación y ejecutaremos como cualquier otra aplicación para poder realizar la instalación. Figura 44.

Figura 44. Creación del instalador.



DISCUSIÓN Y CONCLUSIONES

La construcción de un compilador durante la formación académica de un estudiante, comprendiendo dos semestres en las que se cursan las materias de lenguajes y autómatas I y II es una labor titánica que requiere de mucha dedicación y esfuerzo tanto de parte del estudiante como del profesor, de acuerdo a la experiencia en la construcción de este tipo de software de base, se presenta continuamente la necesidad de buscar herramientas que favorezcan el aprendizaje y que además permitan su construcción de una forma más dinámica. Por lo tanto se puede concluir que la integración de conocimientos en este tipo de proyectos de desarrollo, ocupan fundamentos teórico prácticos desde la herramienta JLEFO 1.0 para la construcción de expresiones regulares y autómatas, el manejo de archivos, la utilización de estructuras de datos, el control de instrucciones de entrada y salida, la validación y asociación de tipos, la jerarquía de operación, el uso de cuádruplos y notaciones infijas y posfijas, y por último la optimización de recursos, así como el uso de herramientas de software para la creación de ejecutables e instaladores. Haciendo hincapié que esto se cursa a la par de la adquisición de conocimientos teóricos y la documentación del mismo proyecto, para entregar así, un producto de software con la calidad requerida. Sin duda alguna, se pone a disposición de los profesores y estudiantes esta información esperando sea de utilidad en este ámbito de desarrollo.

REFERENCIAS BIBLIOGRÁFICAS

Anónimo. (s.f.). tutorialspoint. Obtenido de tutorialspoint: https://www.tutorialspoint.com/es/compiler_design/compiler_design_intermediate_code_generations.htm

Hopcroft, J. E.; Motwani, R.; Ullman, J. D . Introducción a la teoría de autómatas, lenguajes y computación. PEARSON EDUCACIÓN S.A., Madrid, 2007. ISBN: 978-84-7829-088-8. https://www.frlp.utn.edu.ar/materias/sintaxis/automatas-lenguajes-computacion.pdf

Hopcroft, J., Motwani, R., & Ullman, J. (2007) (pp. 61, 62). Introducción a la teoría de autómatas, lenguajes y computación (3a. ed.). Madrid: Pearson Educación.

Ingeniería Informática II26 Procesadores de lenguaje. Estructura de los compiladores e intérpretes. Repositorio de la Universitat Jaume I. Recuperado de: http://repositori.uji.es/xmlui/bitstream/handle/10234/5876/estructura.apun.pdf?sequence=1&isAllowed=y

Java, t. (2018). Conceptos básicos del lenguaje Java. Obtenido de https://www.ibm.com/developerworks/ssa/java/tutorials/j-introtojava1/index.html

López Ismael Jeder Gustavo, Vega Augusto. Análisis y diseño de Algoritmos. ISBN 978-987-23113-9-1 1. Informática. 2. Programación. I. Jeder, Ismael II. Vega, Augusto III. Título CDD 005.1

Márquez, L. (17 de 10 de 2013). Lenguajes y Autómatas II. Obtenido de Análisis

Semántico: http://elvismqz4.blogspot.com/2013/10/analisis-semantico.html

Ontiveros Víctor, Canal de You tube y pdf. Traducción de una expresión en un árbol binario. (Septiembre 2019). Obtenido de Conversión de notación infija a postfija: https://www.infor.uva.es/~cvaca/asigs/AlgInfPost.htm

https://www.youtube.com/watch?v=xtw_d56RXRU&ab_channel=victorontiveros

Ruíz Catalán, Jacinto. Compiladores Teoría e implementación. ISBN: 978-84-937008-9-8. RC Libros.

Salguero Lamillar Francisco José. LÓGICA Y ANÁLISIS DEL LENGUAJE NATURAL. (pp. 1-14). Madrid. Recuperado de: https://idus.us.es/bitstream/handle/11441/70386/Logica%20y%20analisis%20del%20lenguaje.pdf?sequence=1