**Tecnológico de Estudios Superiores de Jocotitlán**

Ingeniería en Sistemas Computacionales

Lenguajes y Autómatas II

Docente:

Dr. en C. Héctor Caballero Hernández

Alumnos:

Julio Ponce Camacho

Adrián Sánchez Villegas

**Documentación Compilador**

**UNIDAD 4**

Grupo: IC-0701

Turno: Matutino

**Índice temático**

[**Índice de tablas** 5](#_Toc90931854)

[**Introducción** 1](#_Toc90931855)

[**Marco teórico** 2](#_Toc90931856)

[**Compilador** 2](#_Toc90931857)

[**Interprete** 2](#_Toc90931858)

[**Gramática** 2](#_Toc90931859)

[**Gramática Libre de Contexto** 2](#_Toc90931860)

[**Símbolos Terminales** 2](#_Toc90931861)

[**Símbolos No Terminales** 2](#_Toc90931862)

[**Símbolo Producción** 3](#_Toc90931863)

[**Árboles de análisis sintáctico** 3](#_Toc90931864)

[**Token** 3](#_Toc90931865)

[**Patrón** 3](#_Toc90931866)

[**Lexema** 3](#_Toc90931867)

[**Fases del compilador** 4](#_Toc90931868)

[**Tabla de Símbolos** 5](#_Toc90931869)

[**Código fuente** 5](#_Toc90931870)

[**Código objeto** 5](#_Toc90931871)

[**Linker o enlazador** 5](#_Toc90931872)

[**Código ejecutable** 5](#_Toc90931873)

[**Desarrollo** 6](#_Toc90931874)

[**Clase Principal** 7](#_Toc90931875)

[Método main 7](#_Toc90931876)

[**Clase Interfaz** 7](#_Toc90931877)

[Metodo Interfaz 8](#_Toc90931878)

[Método ActionPerformed. 8](#_Toc90931879)

[Método Maximizar 10](#_Toc90931880)

[Método Minimizar 11](#_Toc90931881)

[Método CabeceraMouseDragged y Método CabeceraMousePressed 11](#_Toc90931882)

[Método AcercaDe, Debug y manualV 12](#_Toc90931883)

[**Clase Traducción** 12](#_Toc90931884)

[Método comprobadorLexico 13](#_Toc90931885)

[Método comprobadorSintactico 14](#_Toc90931886)

[Método AnalizadorSemantico 16](#_Toc90931887)

[Método Optimizar 16](#_Toc90931888)

[Método CodigoIntermedio 17](#_Toc90931889)

[Método CodigoEnsamblador 18](#_Toc90931890)

[**Clase Ensamblador** 19](#_Toc90931891)

[Método principal 22](#_Toc90931892)

[Método inicializa1 y Método inicializa 22](#_Toc90931893)

[Método paso1 22](#_Toc90931894)

[Método paso2 22](#_Toc90931895)

[Método Abrir 23](#_Toc90931896)

[Método hashtabsim, hash y hashtabop 23](#_Toc90931897)

[Método Consulta y Consulta\_Todo 23](#_Toc90931898)

[Método Cargar y Cargar2 23](#_Toc90931899)

[Método Altatabsim y Altatabop 23](#_Toc90931900)

[Método validatabsim y validadatatabop 24](#_Toc90931901)

[**Conclusiones** 25](#_Toc90931902)

[**Referencias** 26](#_Toc90931903)

[**Anexos** 27](#_Toc90931904)

[**Pruebas funcionales** 27](#_Toc90931905)

[Análisis Léxico: 27](#_Toc90931906)

[Análisis Sintáctico 27](#_Toc90931907)

[Análisis semántico 28](#_Toc90931908)

[Generación de código intermedio 29](#_Toc90931909)

[Optimización de código 29](#_Toc90931910)

[Generación de código ensamblador 30](#_Toc90931911)

[Código Objeto 30](#_Toc90931912)

**Índice de Figuras**

[Figura 1 Estructura de las clases del Software. 6](#_Toc90931913)

[Figura 2 Interfaces que genera la clase Interfaz 8](#_Toc90931914)

[Figura 3 Funcionamiento de los métodos de la cabecera. 11](#_Toc90931915)

[Figura 4 Fases para el uso de los métodos de la clase Traducción 13](#_Toc90931916)

[Figura 5 Funcionamiento de la clase Ensamblador 21](#_Toc90931917)

[Figura 6 Análisis Léxico (Pruebas funcionales). 27](#_Toc90931918)

[Figura 7 Análisis Sintáctico (Pruebas funcionales). 28](#_Toc90931919)

[Figura 8 Análisis semántico (Pruebas Funcionales). 28](#_Toc90931920)

[Figura 9 Código intermedio (Ejecución). 29](#_Toc90931921)

[Figura 10 Código intermedio (Archivo generado). 29](#_Toc90931922)

[Figura 11. Optimización de código (Pruebas funcionales). 30](#_Toc90931923)

[Figura 12. Código ensamblador (Pruebas funcionales). 30](#_Toc90931924)

[Figura 13 Código objeto (Pruebas funcionales). 31](#_Toc90931925)

# **Índice de tablas**

[Tabla 1 Métodos de la clase Interfaz 7](#_Toc90931926)

[Tabla 2 Métodos de la clase Traducción 12](#_Toc90931927)

[Tabla 3 Clases útiles para la clase traducción 19](#_Toc90931928)

[Tabla 4 Métodos de la clase ensamblador 20](#_Toc90931929)

[Tabla 5 Clases de utilidad en la clase Traducción 21](#_Toc90931930)

# **Introducción**

El presente trabajo mostrara una documentación en base al Software del compilador, que contiene cada de las fases del mismo desde el análisis léxico hasta la generación de código.

En el marco teórico se presentará una serie de conceptos que nos ayudaran a entender la teoría del software que se realizara, todo se encuentra de forma ordenada, empezando por los conceptos más básicos como los gramáticos libres de contexto, símbolos terminales, no terminales, hasta llegar los conceptos relacionados al compilador, como las fases del mismo, la tabla de símbolos, el código objeto, código fuente entre otros.

El desarrollo contendrá descripciones sobre el programa, estas mismas harán referencia a ilustraciones numeradas sobre la implementación del código. El programa se describirá a partir de sus clases creadas, se describirá cada una de estas de forma específica, algunas de las más importantes es la clase “Ensamblador” donde se traduce de código fuente a código ensamblador, alguna otra es la clase “Interfaz” que como su nombre lo indica se creó la interfaz del compilador con todas sus opciones y funciones.

En la parte de las conclusiones se darán a conocer los resultados obtenidos sobre el software, así como también algunas dificultades que se tuvieron al implementarlo, después se encuentran las referencias que contiene las fuentes consultadas, y por último estarán los anexos en el que agregaremos salidas de pantalla sobre el funcionamiento del software terminado.

# **Marco teórico**

## **Compilador**

Es un programa traductor cuya función es traducir (compilar) un programa fuente escrito en algún lenguaje de alto nivel a lenguaje máquina. Este programa traducido o programa objeto, normalmente es guardado en memoria secundaria en forma ejecutable y es cargado a memoria principal cada vez que requiera ser ejecutado.

## **Interprete**

Al igual que el compilador, el intérprete traduce un programa fuente escrito en algún lenguaje de alto nivel, pero con la diferencia que cada instrucción es ejecutada inmediatamente, sin generar un programa en lenguaje de máquina.

## **Gramática**

Una gramática ("G") es un conjunto finito de reglas que describen toda la secuencia de símbolos pertenecientes a un lenguaje específico L.

## **Gramática Libre de Contexto**

Estas gramáticas, conocidas también como gramáticas de tipo 2 o gramáticas independientes del contexto, son las que generan los lenguajes libres o independientes del contexto. Los lenguajes libres del contexto son aquellos que pueden ser reconocidos por un autómata de pila determinístico o no determinístico.

Ejemplo:

La siguiente gramática genera las cadenas del lenguaje L1 = {wcwR / w ∈ {a, b}\* } G1 = ({A}, {a, b, c}, P1, S1), y P1 contiene las siguientes producciones:

S1 → A

A → aAa

A → bAb

A → c

## **Símbolos Terminales**

Los terminales son los símbolos básicos a partir de los cuales se forman las cadenas.

## **Símbolos No Terminales**

Los no terminales son variables sintácticas que denotan conjuntos de cadenas. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.

## **Símbolo Producción**

Las producciones de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada producción consiste en:

1. Un no terminal, conocido como encabezado o lado izquierdo de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.
2. El símbolo ->.
3. Un cuerpo o lado derecho, que consiste en cero o más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

## **Árboles de análisis sintáctico**

Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción. El nodo interior se etiqueta con el no terminal A en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta A durante la derivación.

## **Token**

Un token es un par que consiste en un nombre de token y un valor de atributo 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 un identificador. Los nombres de los tokens son los símbolos de entrada que procesa el analizador sintáctico. A partir de este momento, en general escribiremos el nombre de un token en negrita. Con frecuencia nos referiremos a un token por su nombre.

## **Patrón**

Un patrón es una descripción de la forma que pueden tomar los lexemas de un token. En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres que forman la palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja que se relaciona mediante muchas cadenas.

## **Lexema**

Un lexema es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de ese token.

## **Fases del compilador**

* **Análisis de léxico:** A la primera fase de un compilador se le llama análisis de léxico o escaneo. El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas conocidas como lexemas.
* **Análisis Sintáctico:** La segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación.
* **Análisis Semántico:** El analizador semántico utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio.
* **Generación de código intermedio:** En el proceso de traducir un programa fuente a código destino, un compilador puede construir una o más representaciones intermedias, las cuales pueden tener una variedad de formas. Los árboles sintácticos son una forma de representación intermedia; por lo general, se utilizan durante el análisis sintáctico y semántico. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.
* **Optimización código:** La fase de optimización de código independiente de la máquina trata de mejorar el código intermedio, de manera que se produzca un mejor código destino. Por lo general, mejor significa más rápido, pero pueden lograrse otros objetivos, como un código más corto, o un código de destino que consuma menos poder.

Un algoritmo simple de generación de código intermedio, seguido de la optimización de código, es una manera razonable de obtener un buen código de destino.

* **Generación de código:** El generador de código recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código máquina, se seleccionan registros o ubicaciones (localidades) de memoria para cada una de las variables que utiliza el programa. Después, las instrucciones intermedias se traducen en secuencias de instrucciones de máquina que realizan la misma tarea. Un aspecto crucial de la generación de código es la asignación juiciosa de los registros para guardar las variables.

## **Tabla de Símbolos**

La tabla de símbolos es una estructura de datos que contiene un registro para cada nombre de variable, con campos para los atributos del nombre. La estructura de datos debe diseñarse de tal forma que permita al compilador buscar el registro para cada nombre, y almacenar u obtener datos de ese registro con rapidez.

## **Código fuente**

Programa escrito en un lenguaje de alto nivel (texto ordinario que contiene las sentencias del programa en un lenguaje de programación). Necesita ser traducido a código máquina para poder ser ejecutado.

## **Código objeto**

Es el programa fuente traducido (por el compilador) a código máquina. Aún no es directamente ejecutable.

## **Linker o enlazador**

Es el programa encargado de insertar al programa objeto el código máquina de las funciones de las librerías (archivos de biblioteca) usadas en el programa y realizar el proceso de montaje, que producirá un programa ejecutable .exe.

## **Código ejecutable**

Traducción completa a código máquina, realizada por el enlazador, del programa fuente y que ya es directamente ejecutable.

# **Desarrollo**

El compilador desarrollado se programó a partir de un lenguaje de programación de alto nivel, el cual es Java, y en base a su paradigma de programación orientada a objetos en el que está basado se desarrolló el software en diversas clases, tomando como 1 clase para implementar la interfaz de usuario y las demás clases para su funcionamiento lógico. El software está estructurado en las clases que se ven en la Figura 1.

Figura 1 Estructura de las clases del Software.

La clase principal manda a llamar a la clase Interfaz, la cual a base de un constructor implementa cada uno de los componentes gráficos que tiene el Frame principal de la interfaz, apoyándose de la clase Numero de Línea para llevar un conteo de las líneas escritas en el área de texto, y para agregar el contenido a las salidas de la interfaz se manda a llamar la clase traducción donde se realiza el análisis léxico, sintáctico y semántico de lo que ingrese el usuario en el área de texto que está destinada a desarrollar el código fuente del programa, una vez analizada cada una de estas partes, se realiza una optimización de las variables y se genera el código intermedio y el código ensamblador (el cual es hipotético), una vez generado esto se retorna en una variable que manipula la clase interfaz y las almacena en sus respectivos archivos, para al final mandar a llamar a la clase Ensamblador, que realiza un ensamble para generar el código objeto del software, almacenándolo también en su respectivo archivo.

Lo anteriormente mencionado es el funcionamiento general que tiene el software desarrollado, sin embargo, cada uno cuenta con diferentes métodos y algoritmos que realizan funciones diferentes, por lo cual es importante detallarlos a cada uno de ellos por separado, para que de esta manera se pueda comprender de una manera más clara cada uno y entender cómo funcionan de manera conjunta.

## **Clase Principal**

La clase principal en el programa es la que se manda a llamar al inicio de la ejecución, ya que esta cuenta con el método main, y dentro de este método manda a llamar a todos los demás procesos, sin embargo, primero analiza un archivo de configuración, si este no existe crea uno y determina como se presentara visualmente el software, una vez analizado esto, en base a diversos parámetros crea un objeto de tipo interfaz y genera el frame con la configuración esperada.

### Método main

El método main funciona bajo la siguiente estructura de pasos:

1. Inicio
2. Si existe el archivo de configuración, abrir y analizar los parámetros de color.
3. Si no existe el archivo de configuración, crea el archivo y coloca unos por default.
4. Crea un objeto de tipo Interfaz con base en los parámetros de configuración anteriores
5. Configura la apariencia de la interfaz, un tamaño de ventana predefinido, configura los aspectos de decoración de los bordes y lo centra en la ventana.

## **Clase Interfaz**

La clase principal es la encargada de construir toda la interfaz gráfica del software, siendo que define cada uno de los componentes con los que el usuario puede interaccionar. Esta clase hereda de la clase JFrame, por lo que esta clase al convertirse en objeto se puede manejar como Frame, y por lo tanto también implementa escuchadores para determinar cuándo se presionan los botones, ítems o la posición del mouse para mover la ventana. Esta clase está constituida por diversos métodos que cumplen con funciones específicas para lograr el funcionamiento que se espera, los cuales se describen en la tabla 1, determinando cuál es su nombre, los parámetros que utiliza y el valor que retorna.

Tabla Métodos de la clase Interfaz

|  |  |  |
| --- | --- | --- |
| Nombre del método | Parámetros | Valor que retorna |
| Interfaz | Color CabeceraT, Color titulosT,Color titulW, Color textoT, Color FondoT,Color Fondo2T,Color con | void |
| ActionPerformed | ActionEvent e | void |
| Maximizar | Int X, Int Y | Void |
| minimizar | ItemEvent e | Void |
| CabeceraMouseDragged | MouseEvent evt | Void |
| CabeceraMousePressed | MouseEvent evt | Void |
| AcercaDe | Void | JFrame |
| Debug | Void | JFrame |
| manualV | Void | JFrame |

Esta clase tiene el objetivo de generar cada una de las interfaces con las que el usuario puede interaccionar, las cuales se presentan en la figura 2.

Figura 2 Interfaces que genera la clase Interfaz

### Metodo Interfaz

El método es interfaz es un constructor de la clase Interfaz, siendo que cuando se crea un objeto de esta clase, se asigna la configuración de color que tendrá, los cuales son definidos en la clase del método main. En este constructor no se genera algun tipo de algoritmo lógico, únicamente se crean cada uno de los componentes de la interfaz, como lo son los JPanel, JButton, JTextArea, JMenu, JItemMenu, JLabel etc. Se les asigna su apariencia como el color, tamaño, ubicación y se vinculan a sus respectivos listeners para poder programar cual será la función de cada uno de ellos.

### Método ActionPerformed.

El método actionPerformed es el método encargado de mantenerse a la espera de cualquier presionado de botones por parte del usuario. Cuando el usuario presiona un botón este obtiene el código de donde proviene la acción y en base a ello se ejecuta un bloque de instrucciones especifico, destinado a cumplir la función para lo que fue programado el botón. El funcionamiento de este método se rige bajo la siguiente estructura:

1. Inicia
2. Obtiene la fuente del botón presionado
3. Determina que acción ejecutar en base a una serie de comparaciones.
   1. Si el botón presionado fue referente a ejecución o compilación.
      1. Crea un objeto traducción y a base sus métodos generan los archivos de código Intermedio, Ensamblador y seguido con un objeto Ensamblador crea el código Objeto.
   2. Si el botón presionado es referente a Cerrar o Salir
      1. Termina la ejecución del programa.
   3. Si el botón presionado es referente a minimizar
      1. Establece el JFrame a un estado de icono, minimizándola en el sistema.
   4. Si el botón presionado es referente a Cerrar
      1. Termina la ejecución del programa.
   5. Si el botón presionado es referente a abrir
      1. A partir de un JChooser permite seleccionar un archivo dentro del sistema operativo donde se esta ejecutando el programa, si se selecciona un archivo, se obtiene su ruta y su nombre, para después escanear el contenido que tiene dentro y desplegarlo en el área de texto de código de la interfaz.
   6. Si el botón presionado es referente a Guardar
      1. Si el archivo ya existe
         1. Se obtiene la ruta y el nombre del archivo, eliminando su contenido y almacenando el nuevo contenido con base a lo ingresado en el área de código de la interfaz.
      2. Si el archivo no existe
         1. Se usa un JChooser para seleccionar la ubicación del archivo, así como el nombre que se le va a asignar, para que de esta manera en base a esos datos se cree el archivo junto con el contenido que se ingresó en el área de código de la interfaz.
   7. Si el botón presionado es referente a Guardar Como
      1. Se usa un JChooser para seleccionar la ubicación del archivo, así como el nombre que se le va a asignar, para que de esta manera en base a esos datos se cree el archivo junto con el contenido que se ingresó en el área de código de la interfaz.
   8. Si el botón presionado es referente a Nuevo
      1. Se manda a llamar la clase Interfaz a si misma, para de esta manera generar una nueva ventana donde se agregue el nuevo documento.
   9. Si el botón presionado es referente a Optimizar
      1. Crea un objeto de tipo de traducción y con base al método optimizar que posee determina que variables se ocupan y cuales no, mostrando esto en la consola de la interfaz.
   10. Si el botón presionado es referente a Análisis
       1. Crea un objeto de tipo traducción y con base a sus respectivos métodos, realiza un análisis Léxico, Sintáctico y semántico del código ingresado en el área de código de la interfaz, mostrando los errores encontrados o si el código es correcto en la consola de la interfaz.
   11. Si el botón presionado es referente a Temas
       1. Se hace uso del archivo de configuración para ingresar un nuevo parámetro de color de la interfaz a utilizar, en este caso entre claro y obscuro, creando una advertencia para reiniciar el programa para ver los cambios.
   12. Si el botón presionado es referente a Debuger
       1. Se manda a llamar al método Debug() para crear un nuevo JFrame donde se obtengan aspectos referentes al código que está en el área de código, como la lista de tokens, los códigos intermedios y ensamblador que se está generando, las variables que se usan y las que no, y los problemas que este pudiera tener.
   13. Si el botón presionado es referente a Manual
       1. Manda a llamar al método manual para crea un JFrame donde se muestre el manual de usuario
   14. Si el botón presionado es referente a Completa
       1. Manda a llamar al método maximizar para adapatar los componentes a pantalla completa.
   15. Si el botón presionado es referente a Chica
       1. Manda a llamar al método minimizar para adaptar los componentes al tamaño estándar del JFrame.
4. Obtiene el tiempo de ejecución de la instrucción y la muestra en la interfaz.
5. Termina.

### Método Maximizar

Este método tiene el propósito de cambiar el tamaño de cada uno de los componentes de la Interfaz, con el fin de que se adapten al tamaño de pantalla completa de la aplicación. Para realizar este proceso no requiere de un algoritmo como tal, simplemente es una reconfiguración de sus dimensiones y ubicación en los 2 ejes que existen, por lo tanto, requiere de 2 para metros de tipo entero, X y Y, los cuales hacen alusión al alto y ancho de la pantalla; en base a estos valores se realiza un recalculo para reubicar cada uno de los componentes y se adapten correctamente.

### Método Minimizar

Este método tiene el propósito contrario del método maximizar, siendo que lo que realiza es poner el JFrame en un tamaño por defecto, y cada uno de los componentes también se mantienen en un tamaño que se utiliza de manera predeterminada.

### Método CabeceraMouseDragged y Método CabeceraMousePressed

Estos dos métodos tienen relación, ya que trabajan en conjunto para poder realizar el funcionamiento deseado. Estos dos métodos están vinculados a un JLabel en el JFrame principal, el cual hace la función de cabecera en el programa, por lo tanto, estos dos métodos le dan la función como a esta cabecera. Estos dos métodos funcionan de la forma en que se puede ver en la figura 3.



Figura 3 Funcionamiento de los métodos de la cabecera.

Como se ve en la figura 3, es un proceso bastante lineal, el método CabeceraMousePressed se mantiene a la escucha de que se dé clic sobre el componente, cuando se da clic se obtiene la posición del eje X y Y en la que se encuentra el mouse, e inmediatamente se llama al método CabeceraMouseDragged, la cual obtiene la posición actual del Jframe y con base a los valores del cursor mueve la ventana.

### Método AcercaDe, Debug y manualV

Cada uno de estos métodos comparten la característica de que son utilizados para generar nuevas ventanas de interfaz con diferente información, sin embargo, los 3 parten de la misma base, son un método que retorna un JFrame que se configura en el método actionPerformed si el usuario quiere acceder a alguna de estas funciones. Cada uno de estos 3 métodos cuenta con diferentes componentes gráficos con los que puede interaccionar el usuario, el método AcercaDe únicamente genera una ventana donde están los créditos de las personas involucradas en el desarrollo de este proyecto, y el método manualV únicamente genera una pequeña interfaz donde el usuario puede ver cuál es la sintaxis del lenguaje que se utiliza en el compilador. El método debug quizá es el más importante a la hora de realizar cualquier ejecución, ya que este JFrame se divide en varias partes donde se puede observar el código intermedio y ensamblador, una lista de variables clasificadas dependiendo el uso que tengan, los tokens del código y un análisis sintáctico, léxico y semántico del código para determinar si hay errores o algún tipo de problema.

## **Clase Traducción**

Esta es la clase más importante en el programa, ya que en esta es donde se realiza el proceso de compilación, donde se realizan procesos como lo son el análisis léxico, análisis sintáctico, análisis semántico, optimización de código y la generación del código intermedio y ensamblador. Para realizar cada uno de los procesos relacionados a la compilación del programa, se desarrollaron diversos métodos, cada uno con sus diversos procesos y parámetros, estos métodos se pueden ver en la tabla 2.

Tabla Métodos de la clase Traducción

|  |  |  |
| --- | --- | --- |
| Nombre del Método | Parámetros | Valor que retorna |
| ComprobadorLexico | String Codigo | boolean |
| ComprobarSintaxis | String Codigo | boolean |
| eliminar\_ | String res | String |
| revisar | String R[][], String T[][], int col | String |
| AnalizadorSemantico | String[] codigo | boolean |
| optimizar | String[] codigo | String |
| CodigoIntermedio | String Codigo | String |
| generarEtiqueta | -- | String |
| CodigoEnsamblador | String Codigo | String |

La tabla 2 muestra cada uno de los métodos empleados para realizar la traducción del programa a ensamblador y código intermedio, cada uno tiene sus diversos procesos y subprocesos para dar la función deseada, pero la forma en que funcionan en conjunto se puede ver en la figura 4, por lo cual si se desea llegar a una traducción correcta se tienen que mandar a llamar de la forma en cómo se representan en la figura, ya que primero se debe de tener un código fuente, el cual es ingresado en el área de texto predefinida en la interfaz, después se tiene que llamar al analizador léxico, sintáctico, semántico y optimizador, para que de esta manera se pueda realizar la traducción correcta a código intermedio y de ahí al código ensamblador para que esté listo para generar el código objeto. Cabe destacar que las variables más importantes en todo este proceso es la variable TOKENS, la cual funge como una tabla de símbolos, y las variables Generar y error, las cuales se encargan de recopilar cada uno de los errores que se encuentren en cada uno de los análisis que se realicen en el código, siendo uno para almacenar cual es el error y el otro es un contador que determina cuantos errores se han encontrado; estas dos variables son importantes porque se mostraran de salida en la consola, para que el programador detecte cual fue el error encontrado.



Figura 4 Fases para el uso de los métodos de la clase Traducción

### Método comprobadorLexico

Este método tiene el propósito de realizar un análisis léxico del código fuente que ingrese el usuario, siendo que analiza si los caracteres y palabras ingresadas tienen sentido en las reglas establecidas en el software. Este método funciona en base a lo siguiente:

1. Inicia
2. Obtiene el código fuente.
3. Analiza cada uno de los caracteres de las palabras del código.
   1. Si existe el carácter.
      1. Lo almacena en la variable general TOKENS
   2. Si no existe el carácter.
      1. Lo almacena en la variable general TOKENS con estado de desconocido y se agrega el error a la variable General y aumenta el conteo de errores.
4. Realiza una búsqueda si existe el carácter de comillas
   1. Si existe
      1. Realiza un recorrido para encontrar donde cierran, y almacena el valor concatenado como Texto
   2. Si no existe
      1. Continua.
5. Concatena las palabras que están clasificadas como Letras.
6. Determina si son palabras reservadas
   1. Si se encuentra la palabra reservada en la tabla de Tokens
      1. Se clasifica como reservada.
   2. Si no lo encuentra
      1. Se clasifica como variable si son letras y números concatenados.
      2. Se clasifica como numero si son solo números.
      3. Se clasifica como operador si es un operador aritmético o de comparación.
      4. Se clasifica como signo si es un signo de puntuación.
7. Retorna true si no hay errores
8. Fin

Es importante que este método se ejecute al principio para así construir la tabla de símbolos a través de la variable TOKENS, ya que, en base a esta, se realizan los análisis posteriores del código para comprobar si este se implementó correctamente o detectar los errores que se encuentren.

### Método comprobadorSintactico

Este método es el encargado de realizar el análisis sintáctico del código que se ingrese, para que este método se pueda realizar se tiene que realizar previamente el análisis léxico ya que requiere usar la tabla de TOKENS que se genera en ese método. El funcionamiento general de este comprobador Sintáctico, es que con base al código ingresado por el usuario, construye una gramática libre de contexto con los elemento ingresados, para al final hacer una comparación entre la gramática libre de contexto sobre la que se rige el software con la gramática del código que ingreso el usuario, siendo que si no coinciden es porque existe un error, y si coinciden es porque lo ingresado es correcto.

Este método funciona de la siguiente manera:

1. Inicio
2. Obtiene el código fuente ingresado.
3. Construye una gramática haciendo una comparación si existe una gramática que contenga el elemento.
   1. Si contiene el elemento
      1. Analiza los demás elementos siguientes con la misma gramática, y coloca el nombre con el que se clasifico el TOKEN.
   2. Si no contiene el elemento
      1. Comparar con la clasificación del TOKEN
         1. Si no contiene la clasificación
            1. Invocar al método Revisar

Realiza un análisis recursivo para encontrar a que elemento no terminal pertenece.

1. Sustituye los valores de los TOKENS por su clasificación.
2. Compara si la gramática construida existe en las reglas generales
   1. Si existe
      1. Sustituye el fragmento de la gramática por el nombre del elemento no terminal al que pertenece.
   2. Si no existe
      1. Almacena que hay error en la variable General y aumenta el conteo de errores.
3. Concatena la gramática en una variable general llamada Sintáctico.
4. Regresa al paso 3 si faltan líneas del código
5. Compara la gramática general.
   1. Si concuerda
      1. La sintaxis es correcta y retorna true
   2. Si no concuerda
      1. La sintaxis es incorrecta, marca error de sintaxis general y retorna false.
6. Fin

Una vez realizado este proceso de análisis sintáctico puede proseguir a realizar la siguiente fase de compilación, la cual es realizar un análisis semántico para determinar si las variables usadas realmente están declaradas o no. En este método no da como resultado algo que usen los demás métodos para funcionar, sin embargo, es importante que se realice ya que cuando se pasa a las siguientes fases, se da por hecho que la sintaxis es correcta.

### Método AnalizadorSemantico

El método analizadorSemantico tiene el proceso de realizar un análisis semántico del código que ingreso el usuario, pero en este caso es un análisis parcial, ya que únicamente está orientado a determinar si las variables que el usuario está usando están declaradas o sin lo están, haciendo uso de una lista donde se almacenan las variables declaradas y realiza una comparación si la variable de la línea que se analice se encuentra en esa lista o si ya se encuentra declarada previamente.

El funcionamiento de este método es el siguiente:

1. Inicio
2. Obtiene las líneas del código ingresado por el usuario.
3. Analiza cada una de las líneas para determinar si es una declaración.
   1. Si es una declaración
      1. Compara si ya existe en la lista de variables
         1. Si no existe
            1. Almacena el tipo de dato y el nombre de las variables en la lista.
         2. Si existe
            1. Concatena en la variable General que la variable ya esta declarada y aumenta el contador de errores.
   2. Si no es una declaración
      1. Continua
4. Con base a la variable TOKENS busca el nombre de las variables y compara.
   1. Compara con la lista de variables declaradas
      1. Si las variables concuerdan en ambas partes
         1. Continua
      2. Si las variables no concuerdan
         1. Almacena en la variable general que no se declaro la variable y aumenta el contador de errores.
5. Retorna true si no hay errores.
6. Fin.

### Método Optimizar

Este método tiene el propósito de detectar cuales son las variables que están declaradas, pero que no son utilizadas y únicamente pueden ocupar espacio en memoria sin ningún sentido. Este método únicamente avisa al usuario retornando cuales son las variables que no se ocupan, pero el código que le llega al código Intermedio es uno donde ya se han optimizado. Este método funciona de la siguiente manera:

1. Inicio
2. Obtiene las líneas de código
3. Determina cuantas veces aparece cada variable en el código.
   1. Si aparece una vez
      1. Concatena en general que la variable está declarada pero no es utilizada.
   2. Si aparece más de una vez
      1. Concatena la variable en valor donde se almacena el código optimizado
   3. Si no es una variable
      1. Concatena la línea donde se almacena el código almacenado.
4. Actualiza la variable TOKENS
5. Retorna el código optimizado
6. Fin

El código optimizado que retorna este método, será utilizado para la creación del código Intermedio y el código Ensamblador, por lo cual es importante realizarlo. Sin embargo, también se puede hacer uso del código sin optimizar, pero esto no es lo más recomendable, ya que habría uso de memoria poco optimo.

### Método CodigoIntermedio

Este método tiene el propósito de convertir el código optimizado a un código mas simplificado, el cual es el código intermedio, por lo cual lo único que realiza es la conversión de cada una de las instrucciones a un código simple, con uso de etiquetas para realizar saltos condicionales e incondicionales con la instrucción goto. Este método funciona de la siguiente manera:

1. Inicio
2. Obtiene el código optimizado
3. Concatena las cabeceras del programa.
4. Concatena el método principal del programa
5. Determina qué tipo de instrucción es para hacer la conversión
   1. Si es una instrucción de declaración o una asignación
      1. La coloca tal cual, pero omite le punto y coma
   2. Si es una instrucción de salida o entrada de datos
      1. La coloca tal cual, pero omite le punto y coma
   3. Si es una instrucción condicional
      1. Coloca la instrucción y comparación tal cual
      2. Genera una nueva etiqueta y se coloca la instrucción goto hacia la etiqueta.
      3. Coloca la palabra de lo contrario
      4. Genera una nueva etiqueta y se coloca la instrucción goto hacia la etiqueta.
      5. Genera la etiqueta 1 y la instrucción que engloba la etiqueta
      6. Genera la etiqueta 2 y la instrucción que engloba la etiqueta.
   4. Si es una instrucción de tipo For
      1. Genera una etiqueta inicial
      2. Coloca las instrucciones que engloba la etiqueta, con base a los pasos del paso 5 de este algoritmo.
      3. Convierte la instrucción a uno de tipo condicional, siendo que, si el contador llega a la condición, pasa a la siguiente instrucción, en caso que no pasa a la etiqueta inicial.
6. Cierra el paréntesis final
7. Retorna el código Intermedio
8. Fin

Este método genera una traducción del código optimizado a instrucciones de tipo código Intermedio, siendo que este mismo código será utilizado para generar el código ensamblador del código fuente.

### Método CodigoEnsamblador

Este método es el último que se realiza en esta clase de Traducción, siendo que es el encargado de generar el código Ensamblador, que será utilizado para generar el código objeto del software por otra clase. La traducción se realiza de una manera muy similar a la del método para generar el código intermedio, sin embargo, en este caso las instrucciones SI, WRITE, READ, Int, char, float etc. Son sustituidos por sus respectivo nemónicos que den la función que se desea realizar.

Este método funciona de la siguiente manera:

1. Inicio
2. Obtiene el código intermedio
3. Obtiene el nombre del método principal y lo concatena con START
4. Determina qué tipo de instrucción es para hacer la conversión
   1. Si es una instrucción de declaración o una asignación
      1. La establece al inicio, como variables de tipo WORD
   2. Si es una instrucción de salida o entrada de datos
      1. Hace uso de los registros y los nemónicos como LEA.
      2. Hace uso de MOV para generar interrupciones.
   3. Si es una instrucción condicional
      1. Hace una comparación inicial con el nemónico CMP, y a través de saltos condicionales salta a una etiqueta.
      2. Si se cumple la condición se coloca la instrucción JZ, JL etc. hacia la etiqueta.
      3. Sin no se cumple se coloca la instrucción JMP hacia la siguiente etiqueta.
      4. Genera la etiqueta 1 y la instrucción que engloba la etiqueta
      5. Genera la etiqueta 2 y la instrucción que engloba la etiqueta.
   4. Si es una instrucción de tipo For
      1. Genera una etiqueta inicial
      2. Coloca las instrucciones que engloba la etiqueta, con base a los pasos del paso 4 de este algoritmo.
      3. Convierte la instrucción a uno de tipo condicional, siendo que, si el contador llega a la condición, pasa a la siguiente instrucción, en caso que no pasa a la etiqueta inicial con base la comparación de CMP y el salto condicional con JL, JZ etc.
5. Se coloca el nemónico END al final.
6. Retorna el código Intermedio
7. Fin

El código ensamblador que se genera en este apartado, así como el intermedio se almacena en un archivo de texto que será utilizado por otra clase para generar el código objeto en un proceso posterior. La clase traducción hace uso de otras dos clases para funcionar correctamente, las cuales a manera general son las que se pueden ver en la tabla

Tabla Clases útiles para la clase traducción

|  |  |
| --- | --- |
| Nombre de la clase | Función |
| Nodo\_V | Esta clase esta dedicada a la creación de los nodos de una lista que se utilizara para almacenar las variables, con atributos como su nomrbe, tipo de datos y el nodo siguiente al que apunta. |
| Lista | Esta clase esta dedicada a la creación de una lista con uso de los nodos de la clase Nodo\_V, donde se tienen métodos que permiten agregar nodos, quitar nodos, generar código hash, consultar todos los nodos. |

## **Clase Ensamblador**

Esta clase tiene el propósito de tomar el código ensamblador generado por la clase traducción, y realizar una identificación de su estructura, como lo son los símbolos y los operadores, para que una vez identificada cada parte del código se haga una la conversión del código ensamblador a un código objeto (El cual es hipotético, no se pude ejecutar en la maquina debido a la arquitectura diferente del procesador).

Esta clase cuenta con diversos métodos que permiten lograr el funcionamiento deseado en la construcción del código fuente, los cuales se pueden ver en la tabla 3.

Tabla Métodos de la clase ensamblador

|  |  |  |
| --- | --- | --- |
| Nombre del Método | Parámetros | Valor que retorna |
| inicializa1 |  | void |
| inicializa |  | Void |
| principal | String ruta, String nombre | int |
| paso1 |  | Int |
| paso2 |  | Int |
| Abrir | String prog | Void |
| valida | String cad | Int |
| hash | String cad | int |
| Consulta | String z | Int |
| Consulta\_Todo |  | Void |
| valit |  | Void |
| Cargar |  | void |
| Cargar2 |  | Void |
| hashtabsim | String cad | Int |
| Altatabsim | String aa, String bb | Void |
| validatabsim | String cad | Int |
| hashtabop | String cad | Int |
| Altatabop | String cad | Int |
| validadatatabop | String cad | Int |

Dentro de la clase ensamblador, existen diversas subclases que permiten trabajar a partir de estructuras de datos, estas estan presentes en la parte inferior del código. Estas clases son las que se puede ver en la tabla 4.

Tabla Clases de utilidad en la clase Traducción

|  |  |
| --- | --- |
| Nombre de la clase | Utilidad |
| nodo | Crea un nodo en la estructura de datos con atributos como la etiqueta, el operador y el código del operador. |
| tabop | Almacena todos los operadores encontrados, con el nombre del operador y su código. |
| data | Almacena datos referentes al operador y la etiqueta, como lo son el nombre del operador, su código, la etiqueta y su dirección en memoria. |
| tabop2 | Almacena los operadores, pero con la diferencia que también apunta hacia el siguiente operador. |
| tabsim1 | Almacena la tabla de símbolos, con atributos como lo son el nombre de la etiqueta, su dirección y el dato siguiente. |
| nodo1 | Crea un nodo con base al nombre del operador y su código, pero este apunta al siguiente nodo de igual manera. |

La clase ensamblador básicamente funciona de la siguiente manera.



Figura 5 Funcionamiento de la clase Ensamblador

### Método principal

El método principal es el que se manda a llamar para realizar el proceso de construcción del código objeto, donde antes de iniciar detecta si existen los archivos de salida como tabop y tabsim, los cuales son encargados de almacenar los símbolos y operadores, para después mandar a llamar a los paso1 y paso2 para iniciar con el proceso de construcción del código objeto.

### Método inicializa1 y Método inicializa

Estos dos métodos se encargan de inicializar las variables que se utilizan en todo el desarrollo del programa, como lo son las estructuras de datos donde se almacenan los operadores y símbolos, y las diversas variables que se utilizan en la clase, con el fin de que cada que se realice el proceso de la creación del código objeto cada una de las variables a usar estén limpias y no haya problemas con datos basura.

### Método paso1

Este método tiene el propósito de identificar cada uno de los operadores y símbolos que existan dentro del código ensamblador, y de esta manera almacenarlo en las estructuras de datos para poder procesarlos en los demás métodos. La estructura básica que sigue este método es la siguiente:

1. Inicio
2. Obtiene el código ensamblador del archivo ensam.
3. Divide las palabras con base a los espacios.
4. Identifica cada uno de los elementos
   1. Si es un operador
      1. Almacenar en la variable tabop
   2. Si es una etiqueta
      1. Almacenar en la variable Tabsim
   3. Si no
      1. Dice error de operador
      2. Termina
5. Escribe en el archivo ensam1, los elementos identificados
6. Consulta toda la información de los operadores
7. Termina

### Método paso2

En este método se realiza la construcción del código objeto y se escribe en el archivo objeto, tomando como base el tipo de operador, para determinar cuánto espacio en memoria requiere y que valores requiere.

1. Inicio
2. Analiza el archivo ensam1
3. Identifica los operadores
   1. Si es operador de inicio
      1. Se coloca el nombre de la etiqueta principal y donde inicia el programa.
   2. Si es un operador de variable
      1. Se coloca un tamaño especifico.
   3. Si es un operador
      1. Se asigna el valor con base al código del operador
   4. Si es de termino
      1. Se asigna la última dirección
4. Se escriben los valores en el archivo objeto
5. Termina.

### Método Abrir

Este método tiene el propósito de abrir el archivo instrucciones.txt en el cual están todas las instrucciones que soporta para la conversión de código objeto, son alrededor de 60 operadores y cuentan con su identificador que funciona para generar las direcciones de memoria, almacenándose en la estructura de datos tabla, que funciona para validar que existan los operadores que se ingresen en el archivo de ensamblador.

### Método hashtabsim, hash y hashtabop

Estos métodos se encargan de generar un identificador de tipo hash para cada elemento, esto se realiza con base al número de letras que tenga el operador, la cadena o el símbolo que se esté utilizando. Básicamente este proceso se realiza de la siguiente manera:

1. Inicio
2. Analiza cada carácter de la cadena
3. Si es una letra
   1. Se le suma 1 al contador
4. Retorna el contador
5. Fin

### Método Consulta y Consulta\_Todo

Determina si el parámetro que se ingresa existe como un nemónico que viene por defecto, retornando 1 se es que este existe o retornando 0 si no existe, el método consulta todo son solo palabras reservadas específicas y el método de consulta\_Todo hace un análisis del archivo de instrucciones para determinar si el operador analizado existe o no por parte de esas instrucciones.

### Método Cargar y Cargar2

Estos métodos tienen el propósito de realizar la carga de los diferentes datos en cada uno de las estructuras de datos de los operadores, con el fin de hacer uso de ellos para realizar cada uno de los procesos con la información real y de esta manera se genere el archivo objeto y las tablas de símbolos/operadores de manera exitosa y correcta.

### Método Altatabsim y Altatabop

Estos dos métodos se encargan de dar de alta los operadores y símbolos, estos métodos son principalmente utilizados en el paso1, donde se identifican cada uno de los elementos usados en el código ensamblador, para ello primero realiza una comparación si existe el operador o no, y si no existe almacena el valor lo almacena en la estructura para ser utilizado en los demás métodos.

### Método validatabsim y validadatatabop

Estos dos métodos tienen el propósito de validar que los operadores o la tabla de símbolos estén dados de alta en las estructuras de datos de dedicados a almacenar esos valores, con el fin de identificar si se puede hacer uso de ellos en la construcción del código objeto. Estos métodos funcionan básicamente de la siguiente manera:

1. Inicio
2. Analiza el operador o símbolo
3. Obtiene su valor hash
4. Compara si existe con base a su operador hash
   1. Si existe retorna 1
   2. Si no existe retorna 0
5. Termina

# **Conclusiones**

Un compilador es aquel que con base a una serie de reglas gramaticales realiza la conversión de un código fuente escrito en un lenguaje de alto nivel a un código que la maquina sea capaz de interpretar. Para lograr la conversión se requiere de una serie de procesos para lograr efectuar esto, como lo es un análisis Léxico, análisis sintáctico, análisis semántico, optimización de código, generación de código y la generación del código objeto que la computadora sea capaz de interpretar, los análisis se hacen con el fin de que exista coherencia en las instrucciones y se pueda generar algo funcional

En el caso de este proyecto que fue la creación de un compilador con una gramática libre de contexto propia desarrollo gran parte de las fases que tiene un compilador, llevando a cabo un análisis léxico, sintáctico y semántico (Orientado a la declaración de variables) con el fin de determinar que el código que ingrese el usuario sea correcto y pueda seguir con las demás fases, durante los procesos anteriores se creo una variable importante llamada TOKENS, la cual almacena cada uno de los símbolos que se encuentran en el código ingresado, siendo muy importante de modo que las demás fases hacen uso de este para poder realizar los demás análisis, y al igual que esta variable, se hizo uso de otra llamada General para el control de errores, y poder detectar cuando ocurren errores en el código ingresado.

Una vez realizado los análisis de las primeras fases, se realiza la construcción de los códigos intermedio y ensamblador, esto con base a la tabla de TOKENS y con el código fuente que ingrese el usuario, en estas 2 fases teóricamente ya no debería de existir ningún tipo de error, ya que por eso mismo es que se analizó anteriormente el código. Una vez construidos se almacenan en archivos y se realiza la creación del código objeto.

Este proyecto fue un reto debido a la gran cantidad de apartados con las que cuenta un compilador, la mayoría de funciones se realizó de manera correcta, a excepción del manejo de errores, ya que este software desarrollado si marca que existe un error, sin embargo, no de una manera específica, lo cual hace un poco complicado orientar al usuario en base a la detección de los errores, pero fuera de eso la interfaz y el funcionamiento del programa fue el esperado.

## **Pruebas funcionales**

### Análisis Léxico:

Para esta prueba se va a comprobar de que el analizador sintáctico funciona, es decir que se revise la sintaxis de las palabras palabra por palabra, y revisar que estén los símbolos de la misma.

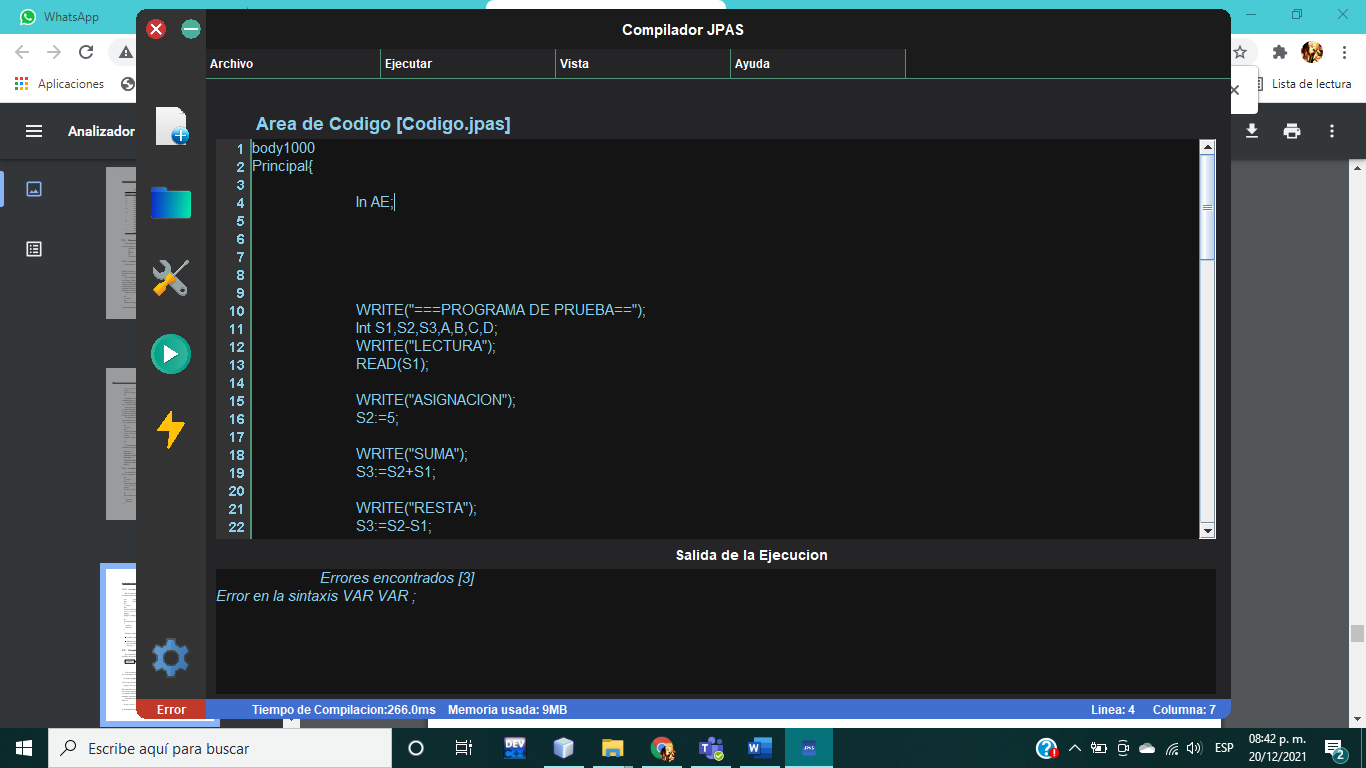


Figura 6 Análisis Léxico (Pruebas funcionales).

Como se puede observar en la Figura 6, el análisis léxico si cumple su función, ya que en este se colocó “In E;” el cual la el tipo de datos está incompleto.

### Análisis Sintáctico

Para esta prueba se escribió una línea completa de código, en este caso se declararon unas variables donde “Z” se encuentra entre comillas simples, es decir que si nuestro análisis sintáctico esta correcto, debe detectar el error de la línea 4. Figura 7.

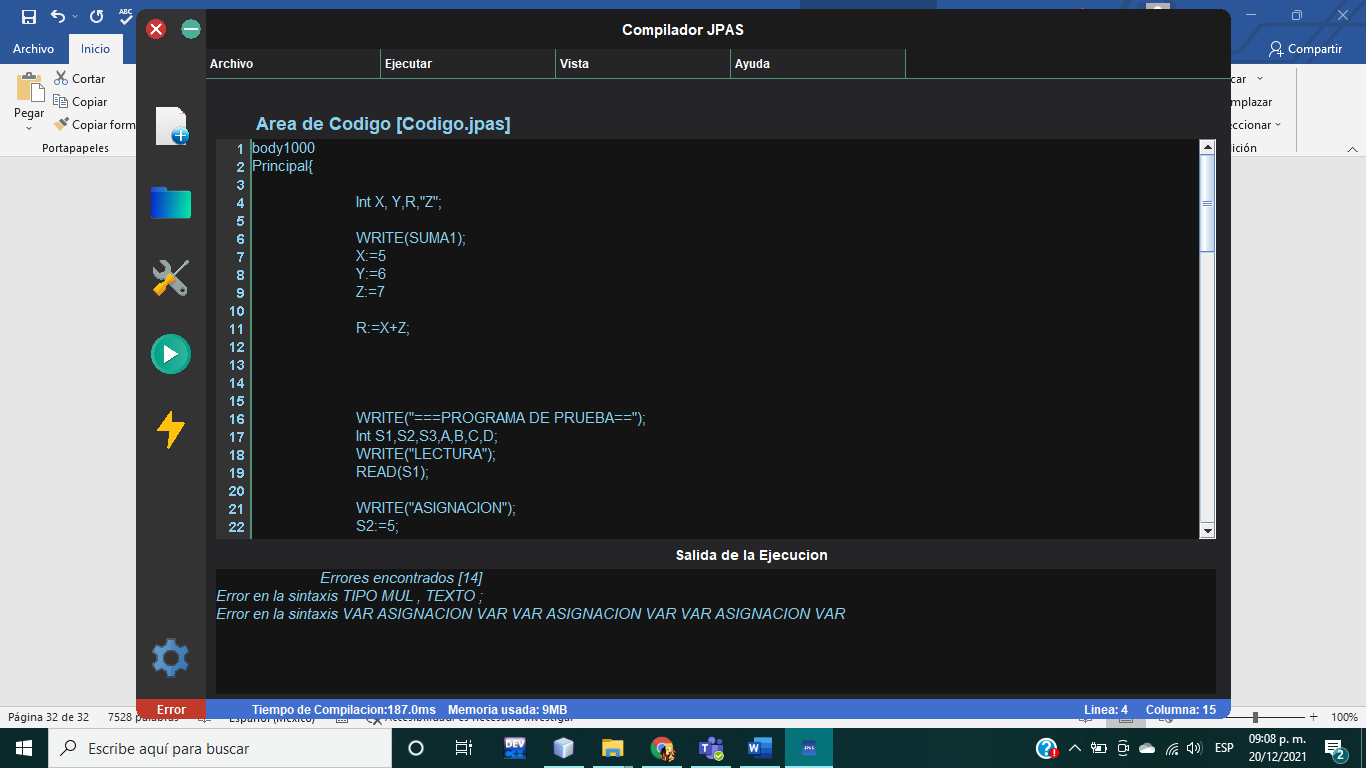


Figura 7 Análisis Sintáctico (Pruebas funcionales).

### Análisis semántico

Aquí se revisa la estructura del programa, por ejemplo, que se haya cerrado una llave, que las variables se encuentren declaradas etc. Para esta prueba se eliminó una de las llaves que cierra al final del código. Figura 8.

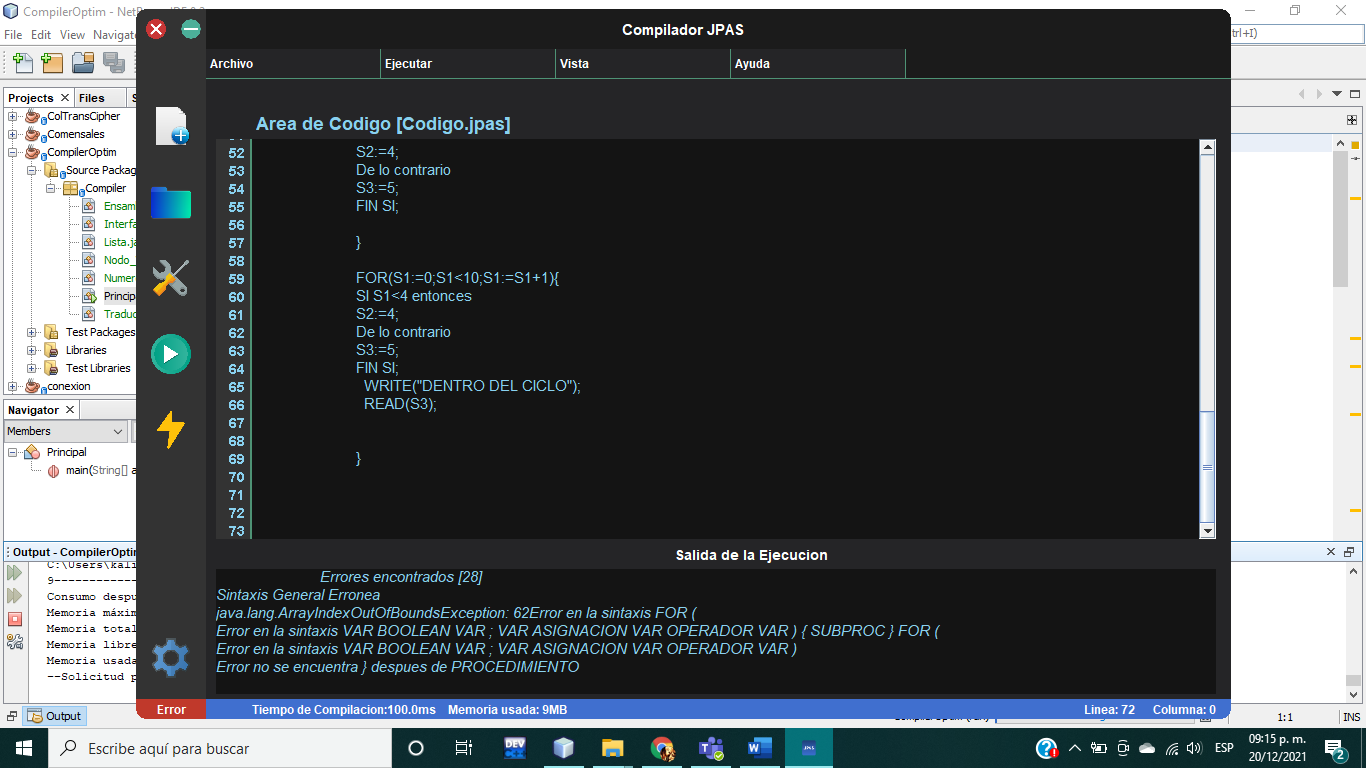


Figura 8 Análisis semántico (Pruebas Funcionales).

Detecta que no se encuentra la llave eliminada después del procedimiento, y esto provoca otros problemas, ya que esta contando la llave del “FOR” como si fuera la principal, entonces se puede decir que si cumple con el análisis semántico.

### Generación de código intermedio

Esta prueba funciona perfectamente, no se deben tener errores léxicos, sintácticos, semánticos, y se debe ejecutar el código, para generar el código intermedio. Figura 9.

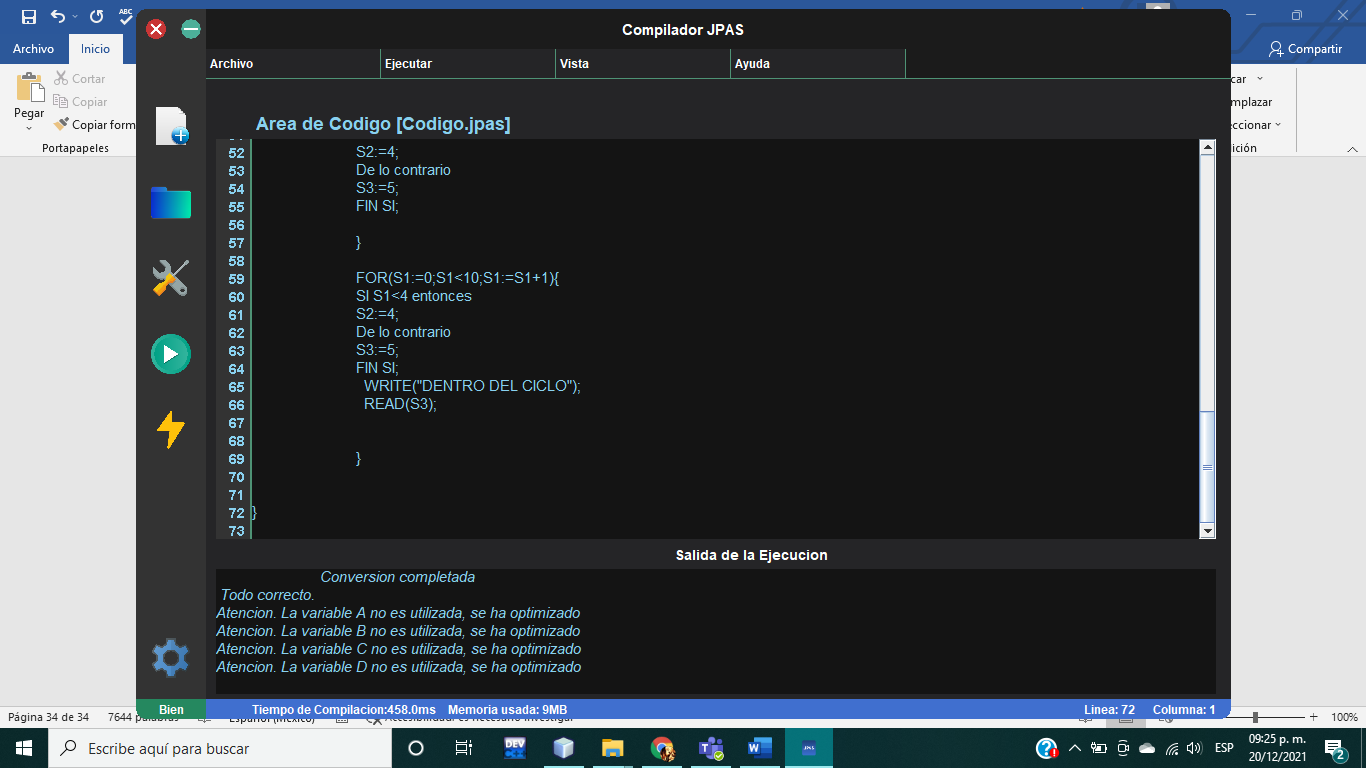


Figura 9 Código intermedio (Ejecución).

El código intermedio se encontrará en la dirección donde se encuentre el archivo jpas. Figura 10.

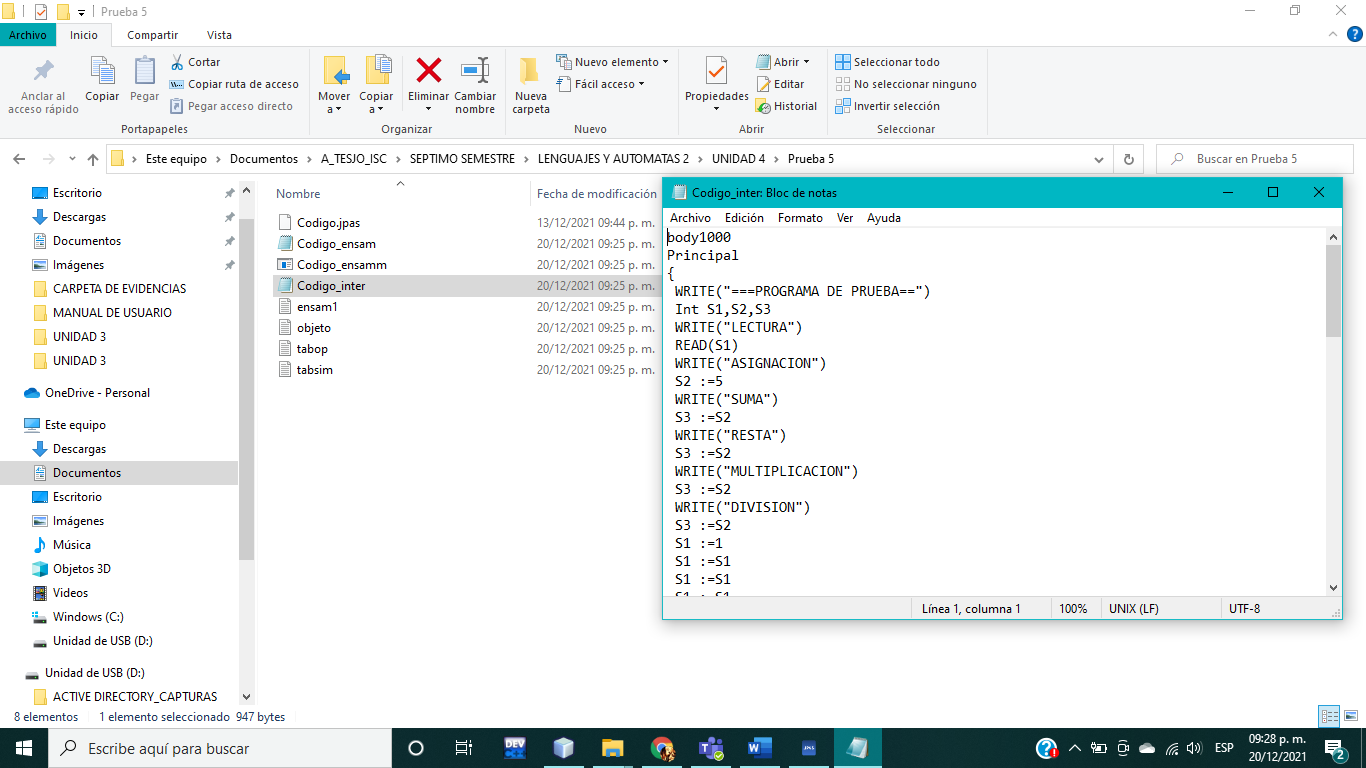


Figura 10 Código intermedio (Archivo generado).

### Optimización de código

Esta prueba es funcional enfocada a mensajes de advertencia sobre variables que no se estén utilizando en el código. Figura 11.

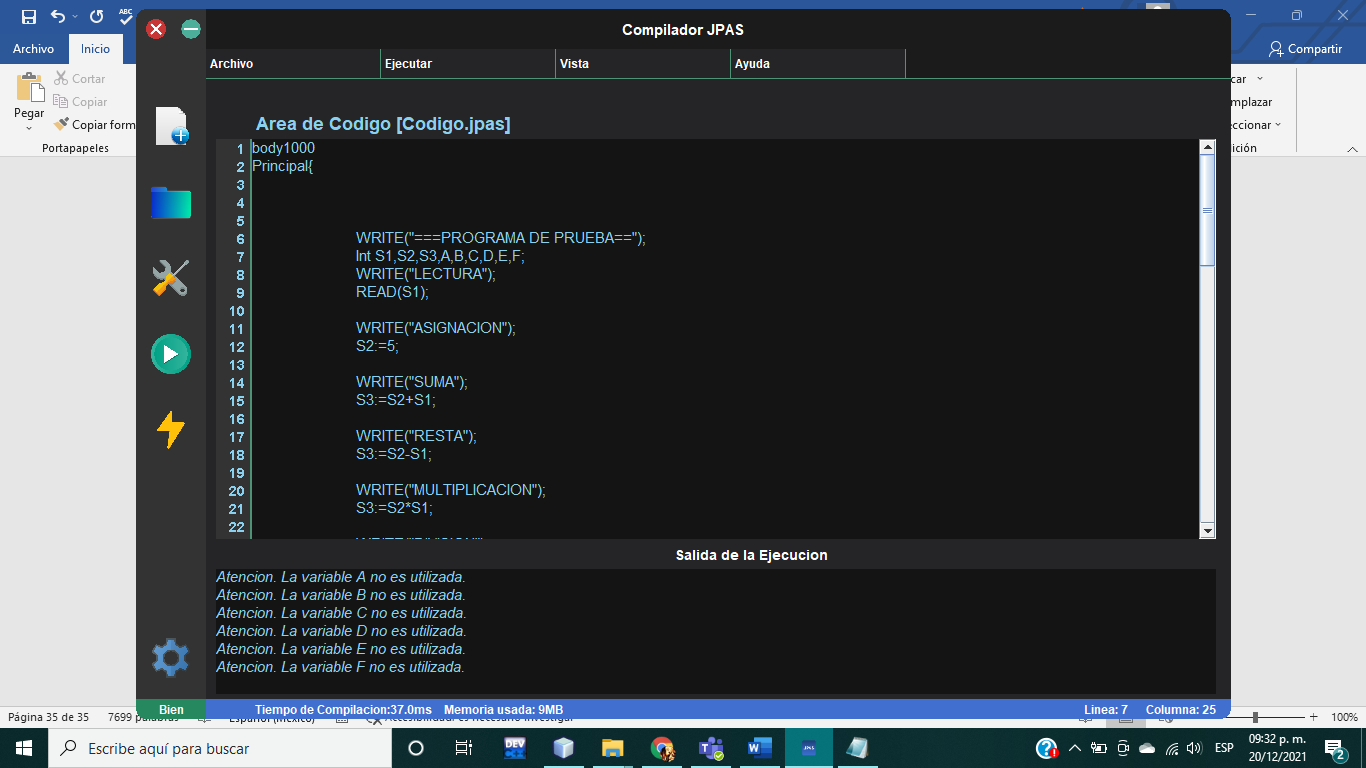


Figura 11. Optimización de código (Pruebas funcionales).

Cabe destacar que, para la generación de código intermedio y ensamblador, estas variables no utilizadas se eliminan, aplicando ahora si la optimización.

### Generación de código ensamblador

Este surge a partir del código intermedio, esta prueba es completamente funcional, este archivo se genera en la dirección donde se encuentre el archivo jpas. Figura 12.

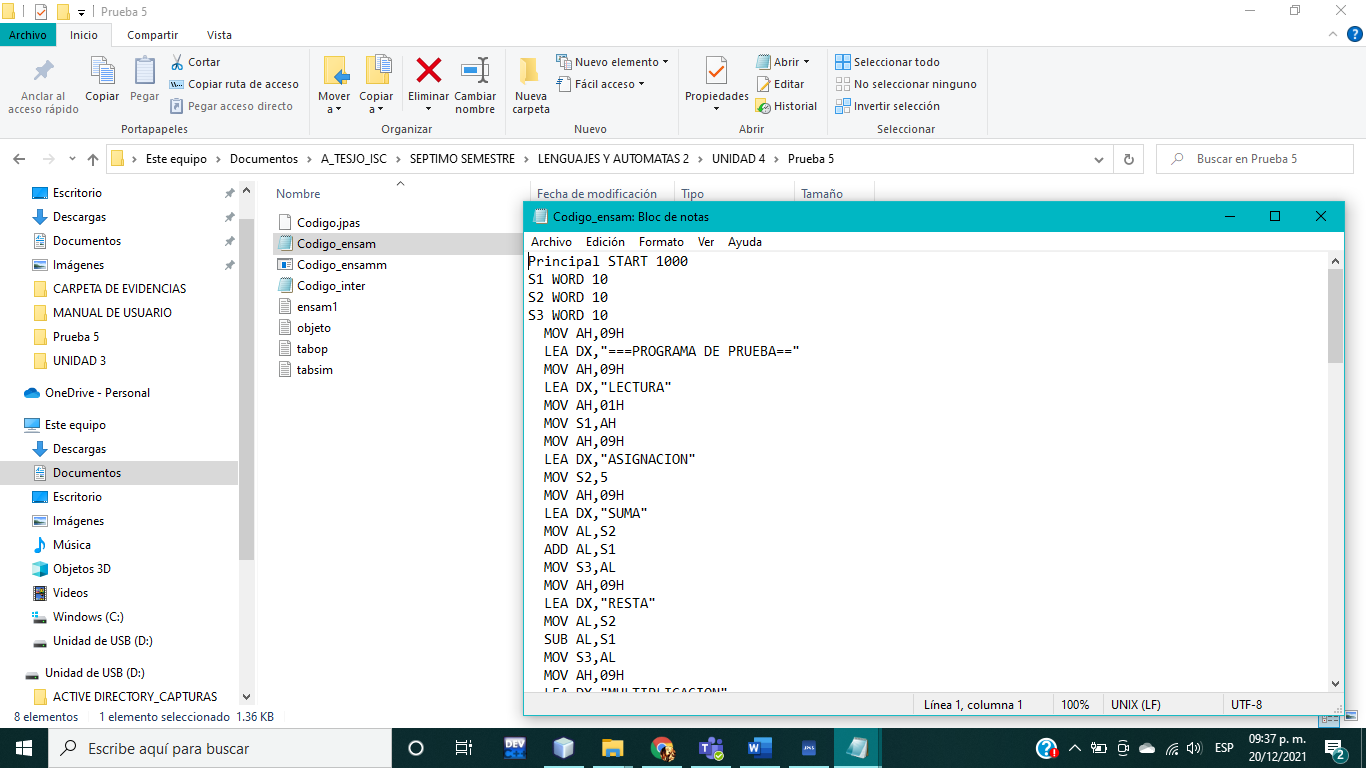


Figura 12. Código ensamblador (Pruebas funcionales).

### Código Objeto

La ultima prueba funcional es la generación del código objeto, el cual se genera en la misma dirección que los anteriores códigos, después de ejecutar el código fuente. Figura 13.

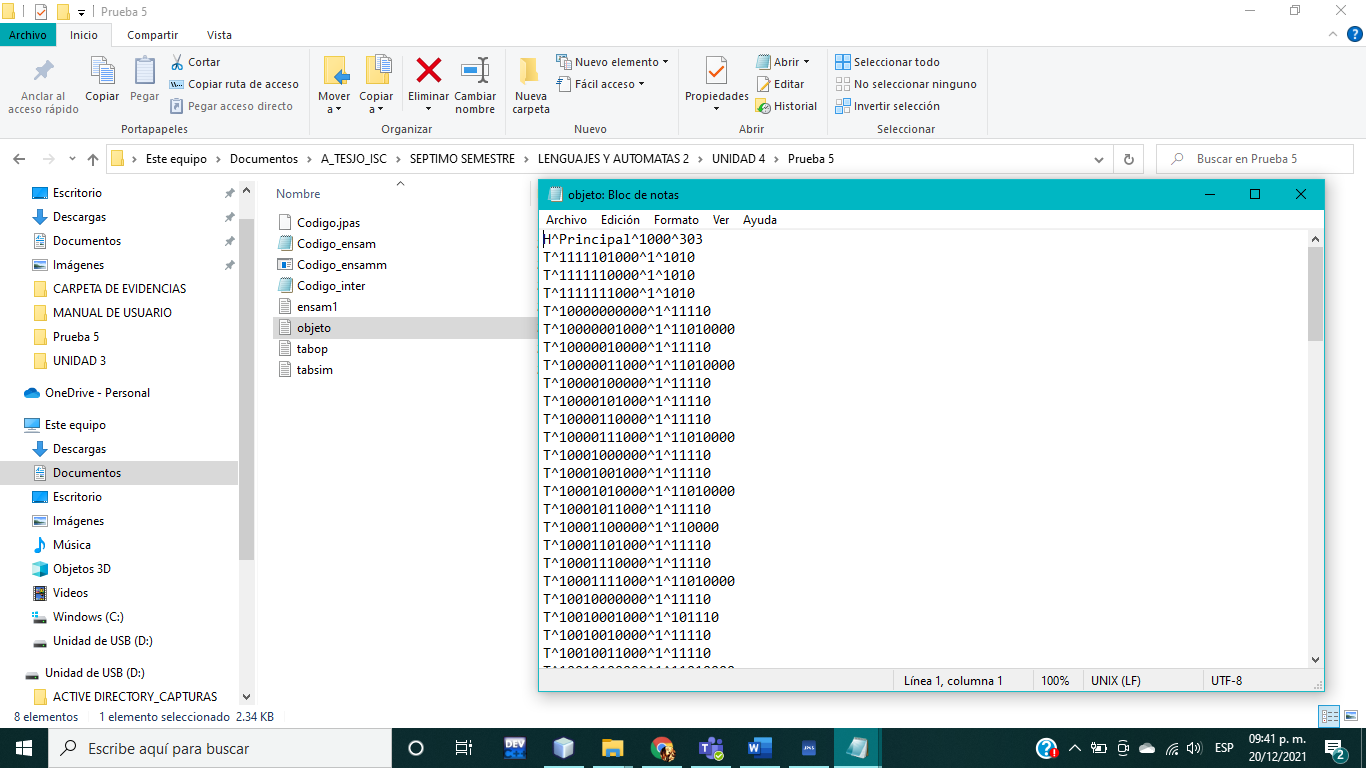


Figura 13 Código objeto (Pruebas funcionales).

Como pudo observar cada una de las pruebas funciono correctamente cumpliendo con las reglas de las fases de compilador que como tal son los requisitos que debe tener este software.

# **Referencias**

1. Alfred V. Aho Mónica S. Lam Ravi Sethi Jeffrey D. Ullman., Compiladores “Principios, técnicas y herramientas" Segunda edición. MÉXICO: PEARSON Educación, 2006.
2. Alfonseca M., De la Cruz M., Ortega A., Pulido E., Compiladores e intérpretes: teoría y práctica, S.A, MADRID: PEARSON Education, 2006.
3. Gramáticas libres del contexto. (2008, noviembre 16) [En Línea]. Disponible en: https://users.exa.unicen.edu.ar/catedras/ccomp1/Apunte5.pdf
4. Fases de un compilador. (2004, Julio 24) [En Línea]. Disponible en: https://kataix.umag.cl/~jaguila/Compilers/T01\_Fases\_Compilador.pdf