

UNIVERSIDAD TÉCNICA DEL NORTE



Facultad de Ingeniería en Ciencias Aplicadas
Carrera de Ingeniería en Sistemas Computacionales

**DESARROLLO DE UN COMPILADOR DE UN LENGUAJE DE PROGRAMACIÓN
ORIGINAL PARA FACILITAR EL PROCESO DE APRENDIZAJE DE
PROGRAMACIÓN EN ESTUDIANTES DE BACHILLERATO**

Trabajo de grado previo a la obtención del título de
Ingeniero en Sistemas Computacionales

Autor:

James Wendell Scarberry

Director:

MSc. José Fernando Garrido Sánchez

Ibarra-Ecuador



UNIVERSIDAD TÉCNICA DEL NORTE

BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago entrega del presente trabajo a la Universidad Técnica el Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a su disposición la siguiente información:

DATOS DE CONTACTO	
CÉDULA DE IDENTIDAD:	175133615-5
APELLIDO Y NOMBRES:	SCARBERRY JAMES WENDELL
DIRECCIÓN:	2260 10 de agosto, Cotacachi, Imbabura
EMAIL:	jwscarberry@utn.edu.ec
TELÉFONO MÓVIL:	099-438-2296


DATOS DE LA OBRA	
TÍTULO:	DESARROLLO DE UN COMPILADOR DE UN LENGUAJE DE PROGRAMACIÓN ORIGINAL PARA FACILITAR EL PROCESO DE APRENDIZAJE DE PROGRAMACIÓN EN ESTUDIANTES DE BACHILLERATO
AUTOR (ES):	SCARBERRY JAMES WENDELL
DIRECCIÓN:	2260 10 de agosto, Cotacachi, Imbabura
FECHA:	19/01/2022
PROGRAMA:	<input checked="" type="checkbox"/> PREGRADO <input type="checkbox"/> POSGRADO
TITULO POR EL QUE OPTA:	INGENIERO EN SISTEMAS COMPUTACIONALES
ASESOR /DIRECTOR:	MSC. FERNANDO GARRIDO

2. CONSTANCIAS

El autor manifiesta que la obra objeto de la presente autorización es original y la desarrolló sin violar los derechos de autor de terceros, por lo tanto, la obra es original y que es el titular de los derechos patrimoniales, por lo que asume la responsabilidad sobre el contenido de la misma y saldrá de la defensa de la Universidad en caso de reclamos por parte de terceros.

Ibarra, a los 11 días del mes de febrero del 2022.

AUTOR:

A handwritten signature in blue ink, reading "James Scarberry", is written over a horizontal black line.

James Wendell Scarberry

CI: 1751336155



UNIVERSIDAD TÉCNICA DEL NORTE
FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

Ibarra, 21 de enero del 2021

CERTIFICACIÓN DEL DIRECTOR

Por medio del presente, yo **MSc. Fernando Garrido**, certifico que el **Sr. James Wendell Scarberry**, portador de la cédula de ciudadanía Nro. **175133615-5** ha trabajado en el desarrollo del proyecto de grado denominado “**DESARROLLO DE UN COMPILADOR DE UN LENGUAJE DE PROGRAMACIÓN ORIGINAL PARA FACILITAR EL PROCESO DE APRENDIZAJE DE PROGRAMACIÓN EN ESTUDIANTES DE BACHILLERATO**”, previo a la obtención del título de Ingeniería en Sistemas Computacionales, lo cual ha realizado en su totalidad con responsabilidad y esmero.

Es todo cuanto puedo certificar en honor a la verdad.

Atentamente,

MSc. Fernando Garrido S. Firmado digitalmente por
MSc. Fernando Garrido S.
Fecha: 2022.02.10
16:17:32 -05'00'

MSc. Fernando Garrido
DIRECTOR DE TESIS

DEDICATORIA

Dedico este trabajo a mi esposa, Daysi Villagómez, por ser el motivo de mi alegría, por inspirarme a ser mejor cada día y por amarme incondicionalmente. Sus palabras de aliento, hermosa sonrisa y cumplidos constantes me impulsaron a siempre seguir adelante y nunca rendirme. Gracias a ella, cada día es una nueva aventura. Si escribiera un libro de mil páginas, no sería suficiente ni para expresar nuestro gran amor, ni para plasmar cuanto la aprecio.

También dedico este trabajo a la pequeña criatura que estamos esperando. Aún no le conocemos, pero le amamos mucho. Gracias por cumplir nuestro sueño.

James Scarberry

AGRADECIMIENTO

Agradezco primeramente a Dios, mi creador y redentor, por darme las capacidades para estudiar esta carrera y realizar este trabajo. Él me sostuvo en todo momento y proveyó cada una de mis necesidades en el momento adecuado. Todo lo que tengo y todo lo que soy es gracias a Él.

Quiero agradecer también a mis padres por haberme enseñado las habilidades académicas básicas que hoy tengo y por haber creído en mí cada paso del camino.

Agradezco a mis suegros por siempre apoyarme y creer en mí.

También doy las gracias a mi cuñado por ser mi mejor amigo y por haber apoyado en este proyecto.

Quiero extender también un profundo agradecimiento a mi tutor, el Msc. Fernando Garrido, por todo el apoyo, paciencia y entusiasmo en esta tesis.

James Scarberry

TABLA DE CONTENIDO

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE	i
DEDICATORIA	iv
AGRADECIMIENTO	v
TABLA DE CONTENIDO	vi
ÍNDICE DE FIGURAS	ix
ÍNDICE DE TABLAS	x
RESUMEN	xii
ABSTRACT	xiii
INTRODUCCIÓN	1
Problema	1
Antecedentes	1
Situación Actual	1
Prospectiva	1
Planteamiento del Problema	2
Objetivos	3
Objetivo General	3
Objetivos Específicos	3
Alcance	3
Metodología	3
Justificación	4
Justificación Educativa	4
Justificación Tecnológica	4
CAPÍTULO 1	5
1.1 - Evolución de los lenguajes de programación	5
1.1.1 – Programación a bajo nivel	5
1.1.2 – Programación a nivel medio	6
1.1.3 – Programación a alto nivel	7
1.1.4 - ¿Por qué existen muchos lenguajes de programación?	11
1.2 – Compiladores y sus Fases	12
1.2.1 - Compiladores: Conceptos Básicos	12
1.2.2 - Análisis Léxico	13
1.2.3 - Análisis Sintáctico	17
1.2.4 - Análisis Semántico	22
1.2.5 - Generación de Código Intermedio	25

1.2.6 - Generación de Código Destino	28
1.2.7 - Optimización de Código	28
1.2.8 - Compiladores de Lenguajes Naturales	30
1.3 - La Programación en la Educación	30
1.3.1 - Programación a nivel de educación básica y bachillerato	30
1.3.2 - Programación a nivel universitario	33
1.3.3 - Cerrando la brecha de género en la programación	33
CAPÍTULO 2	35
2.1 - Introducción	35
2.2 – Recursos y Materiales	35
2.2.1 - Materiales	35
2.2.2 - Software Utilizado.....	36
2.3 – Metodología	36
2.4 – Especificación del lenguaje	42
2.5. – Desarrollo de la Interfaz Gráfica	43
2.5.1 – Arquitectura de la Interfaz Gráfica.....	43
2.5.2 – Apariencia y Funcionalidades Básicas	43
2.5.3 – Funcionalidades Adicionales	47
2.6 – Desarrollo del Analizador Léxico (Explorador)	48
2.6.1 – Arquitectura del Analizador Léxico	48
2.6.2 – Creación y Optimización del Autómata Finito Determinista	48
2.6.3 – Funcionamiento del Analizador Léxico	50
2.7 – Desarrollo del Analizador Sintáctico (Reconocedor)	51
2.7.1 – Arquitectura del Analizador Sintáctico	51
2.7.2 – Creación del Autómata de Pila.....	52
2.7.3 – Funcionamiento del Analizador Sintáctico.....	52
2.8 – Desarrollo del Analizador Semántico	53
2.8.1 – Arquitectura del Analizador Semántico	53
2.8.2 – Funcionamiento del Analizador Semántico	53
2.9 – Ejecutor	54
2.10 – Pruebas	55
2.11 – Recopilación de Código Compilado.....	55
2.12 – Método de Actualización Automática.....	57
CAPÍTULO 3	58
3.1 – Introducción a los Resultados	58
3.1.1 – Alcance	58

3.1.2 – Categoría de Usuarios	58
3.1.3 – Datos de la Encuesta	59
3.1.4 – Descripción de Resultados	60
3.2 – Medida de la Eficiencia de Desempeño (ISO 25010:2011)	60
3.2.1 – Comportamiento Temporal	60
3.2.2 – Utilización de Recursos - RAM	62
3.2.3 – Utilización de Recursos - Procesador	63
3.3 – Medida de la Adecuación Funcional (ISO 25010:2011)	64
3.3.1 – Completitud Funcional	64
3.3.2 – Corrección Funcional	66
3.3.3 – Pertinencia Funcional	67
3.4 – Análisis de Datos Adicionales	71
3.4.1 – Encuesta	71
3.4.2 – Compilaciones de Código	74
3.5 – Conclusión de Resultados	78
CONCLUSIONES	80
RECOMENDACIONES	82
REFERENCIAS	83
ANEXOS	86
Anexo A: Lexemas reconocidos por el analizador léxico	86
Anexo B: Palabras reservadas	88
Anexo C: Funciones preprogramadas	90
Anexo D: Reglas gramaticales reconocidas por el analizador sintáctico	92
Anexo E: Reglas y validaciones del analizador semántico	95
Anexo F: Operaciones permitidas en el lenguaje	101
Anexo G: Lista de errores	102
Anexo H: Errores detectados durante el uso del compilador	104
Anexo I: Código fuente del Compilador Cóndor	106

ÍNDICE DE FIGURAS

Fig. 1. Árbol de Problemas.	2
Fig. 2. Entradas y salidas para cada proceso.	4
Fig. 3. Árboles de derivación que muestran ambigüedad tras tener dos posibilidades con un solo enunciado: id + id * id.	18
Fig. 4. Árbol de derivación: $a = 5 + 3$	23
Fig. 5. Arquitectura del programa con entradas, salidas, controles y mecanismos.	37
Fig. 6. Interconexión de los diferentes componentes del programa y sus entradas, salidas, controles y mecanismos.	38
Fig. 7. Leyenda de símbolos usados en los diagramas de flujo de información.	39
Fig. 8. Diagrama de flujo de información durante el proceso de actualización.	39
Fig. 9. Diagrama de flujo de información durante el proceso de envío de datos a la base de datos. .	39
Fig. 10. Diagrama de flujo de información durante el proceso de lectura y escritura de archivos de código fuente.	40
Fig. 11. Diagrama de flujo de información durante el proceso de análisis de código.	40
Fig. 12. Diagrama de flujo de información durante el proceso de compilación.	41
Fig. 13. Arquitectura de la interfaz gráfica.	43
Fig. 14. Pantalla de bienvenida del compilador donde se indica su versión, creador, asesor y la versión.	44
Fig. 15. Interfaz del compilador.	44
Fig. 16. Menú de funciones desplegadas tras seleccionar "Archivo" y funciones principales.	45
Fig. 17. Menú de funciones desplegadas tras seleccionar "Opciones".	46
Fig. 18. Menú de funciones desplegadas tras seleccionar "Ayuda".	46
Fig. 19. Tabla de errores del compilador.	46
Fig. 20. Arquitectura del Analizador Léxico del compilador.	48
Fig. 21. Tabla de transiciones del AFD generada por Sefalas.	48
Fig. 22. Tabla de transiciones pasado a Excel tras reducir las entradas de 104 a 33.	49
Fig. 23. Tabla de transiciones tras reducir los estados de 36 a 16.	49
Fig. 24. Tabla de transiciones tras agregar la entrada 'λ' y reenumerar los estados.	50
Fig. 25. Arquitectura del analizador sintáctico del compilador.	51
Fig. 26. Arquitectura del analizador semántico del compilador.	53
Fig. 27. Arquitectura del ejecutor del compilador.	55
Fig. 28. Tablas de la base de datos usada para registrar usuarios, código compilado y errores detectados.	56
Fig. 29. Arquitectura de la conexión a la base de datos.	56
Fig. 30. Diagrama porcentual de tipos de usuarios que llenaron la encuesta.	58
Fig. 31. Diagrama de usuarios que han programado antes de la utilización del compilador.	59
Fig. 32. Comparación entre respuesta de tiempo en ambas computadoras durante ejecución normal.	61
Fig. 33. Comparación porcentual de compilaciones exitosas con las falladas.	77
Fig. 34. Gráfica del modelo de regresión lineal entre porcentaje de errores vs compilaciones de los usuarios.	77
Fig. 35. Comparación porcentual de tipos de errores detectados.	78
Fig. 36. Diagrama de los errores semánticos detectados.	105

ÍNDICE DE TABLAS

TABLA 1.1 Tabla de análisis sintáctico LL(1).....	20
TABLA 1.2 Tabla de análisis sintáctico LR.....	21
TABLA 1.3 Representación de código de tres direcciones mediante cuádruplos: $a = 5$, $b = 2$, $c = a + b$	26
TABLA 1.4 Representación de código de tres direcciones mediante tripletas: $a = 5$, $b = 2$, $c = a + b$. .	26
TABLA 3.1 Datos estadísticos del tiempo que los usuarios se tomaron para llenar la encuesta.	59
TABLA 3.2 Datos informativos de los tiempos de carga y procesamiento en la computadora de prueba A.....	60
TABLA 3.3 Datos informativos de los tiempos de carga y procesamiento en la computadora de prueba B.....	61
TABLA 3.4 Datos estadísticos del rendimiento del compilador según la encuesta realizada.....	62
TABLA 3.5 Datos informativos sobre la utilización de memoria RAM para diferentes etapas de funcionamiento del compilador.....	63
TABLA 3.6 Datos comparativos de la utilización de recursos del procesador en las computadoras de prueba.....	63
TABLA 3.7 Datos estadísticos sobre el criterio de los usuarios sobre poder usar el compilador como primer lenguaje de programación.	66
TABLA 3.8 Datos estadísticos sobre el cumplimiento de las expectativas del compilador a los usuarios.....	66
TABLA 3.9 Datos estadísticos sobre la exactitud del compilador a criterio de los usuarios.....	67
TABLA 3.10 Datos estadísticos sobre la facilidad de las palabras reservadas del compilador según los usuarios.....	68
TABLA 3.11 Datos estadísticos sobre la facilidad de la sintaxis según el criterio de los usuarios del compilador.	68
TABLA 3.12 Datos estadísticos sobre la efectividad de los manuales proporcionados a los usuarios del compilador.	69
TABLA 3.13 Datos estadísticos sobre la efectividad de los colores usados para el código fuente.....	69
TABLA 3.14 Datos estadísticos sobre la efectividad de la tabla de errores proporcionada en el compilador según los usuarios.....	70
TABLA 3.15 Datos estadísticos sobre la efectividad de la opción de inserción rápida de bucles y estructuras de control.....	70
TABLA 3.16 Datos estadísticos sobre la primera impresión de los usuarios acerca de la programación.	71
TABLA 3.17 Datos sobre el interés de los usuarios para continuar con una carrera relacionada con la programación.....	71
TABLA 3.18 Datos estadísticos sobre la facilidad de la programación con palabras reservadas en español.....	72
TABLA 3.19 Datos estadísticos de la facilidad del sintaxis del lenguaje para nuevos estudiantes.....	72
TABLA 3.20 Datos estadísticos sobre la experiencia de los usuarios del compilador durante su utilización.	73
TABLA 3.21 Datos estadísticos sobre los criterios de las características de multimedia que ofrece el compilador a los usuarios.	73
TABLA 3.22 Datos estadísticos del criterio de los usuarios sobre si la posibilidad de que las características multimedia generaren un mayor interés a nuevos estudiantes de carreras relacionadas con la programación.....	74

TABLA 3.23 Datos estadísticos sobre si el compilador es recomendable para los nuevos usuarios. ...	74
TABLA 3.24 Datos estadísticos de la cantidad de compilaciones de los usuarios del compilador.	75
TABLA 3.25 Datos estadísticos sobre la cantidad de equivocaciones de los usuarios al compilar sus códigos.	75
TABLA 3.26 Datos estadísticos sobre la cantidad de errores por compilación de los usuarios.....	76
TABLA 3.27 Tabla de frecuencia sobre la cantidad de errores detectados en cada compilación.	76
TABLA 3.28 Tabla de frecuencias de los tipos de errores detectados.	78
TABLA 4.1 Lista de lexemas reconocidos en el lenguaje.....	86
TABLA 4.2 Lista de palabras reservadas.....	88
TABLA 4.3 Lista de funciones preprogramadas.	90
TABLA 4.4 Lista de reglas gramaticales.....	92
TABLA 4.5 Lista de reglas y validaciones semánticas.....	95
TABLA 4.6 Operaciones permitidas.....	101
TABLA 4.7 Lista de errores detectables por el compilador.....	102
TABLA 4.8 Tabla de los errores sintácticos detectados.	104
TABLA 4.9 Tabla frecuencia de los errores semánticos detectados.	105

RESUMEN

Se cree que los estudiantes de bachillerato en Ecuador podrían captar mejor la materia de programación aprendiéndola en su lengua materna usando palabras en español. Viendo la alta tasa de estudiantes que repiten las materias de programación o que salen de las carreras de Ingeniería en Sistemas Computacionales e Ingeniería en Software en la FICA de la UTN, se ha llegado a la conclusión la necesidad de mayor comprensión previa sobre estos temas de programación y aprendizaje de lenguajes formales.

Para cumplir con esto, tras una revisión profunda de literatura sobre la temática, se ha propuesto crear un compilador desarrollado para un nuevo lenguaje con un léxico y gramática sencillos. Las palabras reservadas del léxico son en español para eliminar la barrera de lenguaje durante el aprendizaje inicial. También hay funcionalidades multimediales para aumentar el interés y cerrar la brecha de género que existe en el campo informático. El compilador fue evaluado mediante pruebas de usuario y encuestas.

De las personas que probaron el compilador, hubo 53% de compilaciones acertadas. La eficiencia de desempeño y adecuación funcional, como características de calidad del compilador, muestran niveles tolerables. Según los datos de la encuesta, el usar el Compilador Cóndor puede servir como punto de partida en el aprendizaje de la programación e incluso aumentar el interés en ella.

Palabras claves: compilador, autómata, lenguaje formal, código, multimedia

ABSTRACT

It is thought that high-school students in Ecuador can understand the subject of programming better when learning it in their native language, in this case using words in Spanish. With the high rate of students that fail programming courses or that drop out of the Computer Systems Engineering and Software Engineering majors in the FICA faculty of the UTN university, it has been concluded that there is a necessity of a better previous understanding of these subjects (programming and formal language theory).

To accomplish this goal, after extensive literature review, a compiler was developed for a new programming language with a simple lexicon and grammar. The lexicon's reserved words are in Spanish to eliminate the language barrier during the initial learning process. There are also multimedia functionalities to increase interest and close the gender gap that exists in the computer science field. The compiler was evaluated by user tests and surveys.

Among the people that tried the compiler, 53% of the compilations were correct. The quality characteristics of the compiler, efficiency and functional correctness, were within tolerable levels. According to the survey's data, using Condor Compiler could serve as a starting point in learning programming and even to increase the interest in it.

Keywords: compiler, automaton, formal language, code, multimedia

INTRODUCCIÓN

Problema

Antecedentes

Durante las primeras seis décadas del desarrollo y uso de las computadoras modernas, se vio enormes transformaciones en el aspecto de la tecnología y en los lenguajes usados para programarlas. Durante este desarrollo, se ha ido facilitando el proceso de escribir código y, por ende, el aprendizaje de la programación. Cada nueva generación de lenguajes busca ser más natural y parecido a lenguajes naturales.

Tomando en cuenta esta información, no queda fuera de debate la debida existencia de lenguajes de programación con palabras reservadas en más lenguajes naturales. Sin embargo, casi todos los lenguajes de programación relevantes tienen palabras reservadas exclusivamente en inglés. Aunque, con el tiempo, el uso de las palabras reservadas se vuelve automático, se debe tomar en consideración que el aprendizaje puede ser más sencillo y ágil empezando con un lenguaje de programación que utilice palabras reservadas en la lengua materna. Actualmente, no se cuenta con herramientas relevantes con esta característica. Hay muchos estudiantes en diferentes lugares del mundo que tienen dificultades y retrasos al momento de aprender la programación por este motivo.

Situación Actual

En la Universidad Técnica del Norte (UTN), existe un nivel excesivo de repitencia en las materias de programación en todas las carreras de la Facultad de Ingenierías de Ciencias Aplicadas (FICA). Este hecho existe, a pesar de la gran cantidad de alumnos que ingresan tras haber obtenido un título de bachillerato técnico informático. Muchos de los estudiantes se retiran por su carencia de interés y entendimiento en dichas materias. Otros se gradúan sin haber obtenido una competencia adecuada en ellas.

Otra consideración que se debe tomar en cuenta es la disparidad de género que existe en la FICA, y en particular, las carreras de Ingeniería en Sistemas Computacionales (antigua) e Ingeniería en Software (nueva). Hay muchas más personas del género masculino que del género femenino dentro de dichas carreras.

Prospectiva

Viendo la alta tasa de repitencia y bajo rendimiento en las materias de programación, se propone ayudar a aliviar estos problemas mediante la creación de un compilador y su respectivo lenguaje de programación con un léxico en español. Esto permitirá a los

estudiantes aprender la programación en su lengua materna, sin necesidad de contar con conocimientos previos en inglés. El lenguaje también contará con aspectos multimediales para aumentar el interés en el área.

Planteamiento del Problema

Las unidades educativas de educación secundaria que enseñan informática en el país suelen enseñar lenguajes de programación con palabras reservadas en inglés. Esto genera cierta confusión para los estudiantes no versados en el inglés, y reduce la cantidad de conocimientos grabados por la brecha del idioma. Además, influye en el posterior interés que tendrá el o la estudiante de seguir una carrera profesional basado en la programación. La figura 1 muestra esta relación de las causas y los efectos del problema.

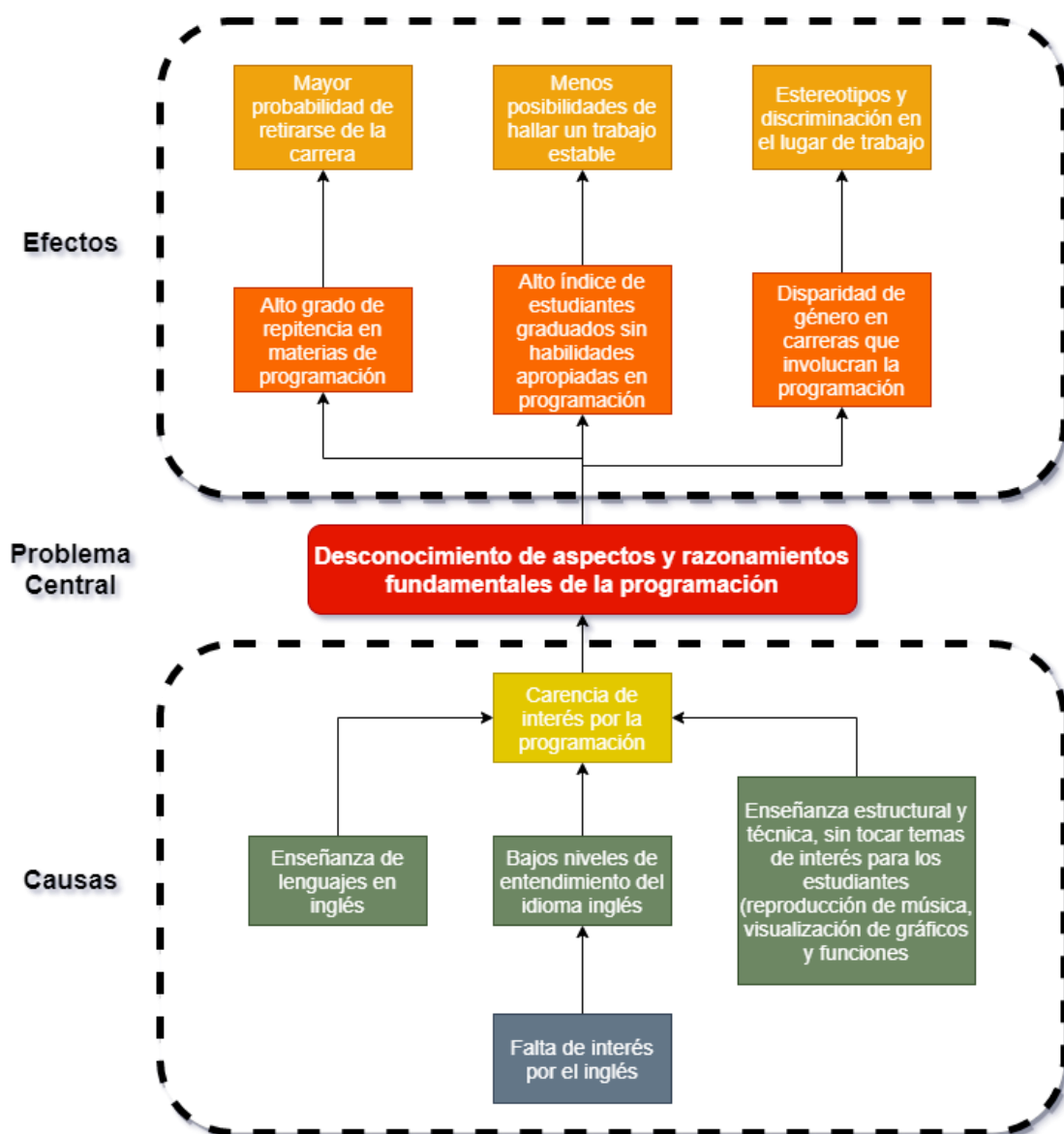


Fig. 1. Árbol de Problemas.

Objetivos

Objetivo General

Desarrollar un compilador de un lenguaje de programación original para facilitar el proceso de aprendizaje de programación en estudiantes de bachillerato

Objetivos Específicos

- Estudiar las teorías sobre la construcción de compiladores
- Crear un lenguaje de programación propio con un alfabeto en español y su respectivo compilador
- Validar el software con las subcaracterísticas Eficiencia de Desempeño y Adecuación Funcional del ISO 25010:2011

Alcance

Desarrollo de un lenguaje de procedimientos y su respectivo compilador en C#, para compilar código del nuevo lenguaje a C# y a un archivo ejecutable. El lenguaje implementará el framework XNA, permitiendo el despliegue de gráficos de funciones matemáticas. Además, se podrá interactuar con una consola. El lenguaje tendrá aspectos multimediales, en cuanto al poder reproducir sonidos y dibujar gráficos importados. Las palabras reservadas serán en español. El lenguaje de programación original que se creará tendrá las siguientes características: las palabras reservadas serán en español, los identificadores tendrán el formato de identificadores en C#, tendrá definición de tipos comunes (entero, real, flotante, booleano, cadena de caracteres, carácter), se podrá declarar funciones, contendrá las estructuras de control de secuencia (condicionales, bucles, repeticiones, etc.), tendrá aspectos multimediales, en cuanto al poder reproducir sonidos y dibujar gráficos importados. Muchos aspectos del lenguaje serán parecidos al lenguaje C#. El compilador recibirá un archivo fuente de extensión .sms (Spanish Multimedia Source) y generará un archivo .exe. Este compilador será compatible en equipos con el procesador Intel x64 y con el framework .Net V. 4.5.

Metodología

El tipo de investigación a realizar será de finalidad aplicada, diseño no experimental, enfoque cualitativo y con fuente de datos documentales. Se desea construir un compilador de acuerdo con teorías de compiladores y usando un conjunto de técnicas de diseño que permitan el diseño de un lenguaje fácil de aprender para estudiantes de habla española, pero potente en manejo de aspectos multimediales. La metodología Cajas SAD (System Architecture Design) se utilizará para el desarrollo del compilador. Esta metodología de cajas

transparentes y negras es un sistema autoorganizado, que analiza las entradas, los procesos sobre ellas y las salidas de información. Implica etapas de Problema, Divergencia, Transformación, Convergencia y finalmente Evaluación. La guía MISRA-C (con las adaptaciones necesarias para C#) se usará para la construcción de código más fiable y seguro. Se evaluará al final la eficiencia de desempeño, según el ISO 25010:2011, mediante una serie de pruebas, y la adecuación funcional mediante encuestas a estudiantes y profesores de informática. Se puede evidenciar un resumen de todo el proceso a seguir en la figura 2.



Fig. 2. Entradas y salidas para cada proceso.

Justificación

Justificación Educativa

Aportar al ODS #4 de la educación, mediante la posibilidad del aumento de estudiantes que completen los estudios universitarios, el acrecentamiento de competencias profesionales de graduados y un decremento en el desequilibrio que existe de estudios ingenieriles entre el género masculino y el femenino. Se espera poder contribuir a estas metas mediante un lenguaje de programación en la lengua materna de estudiantes de bachillerato, con capacidades de interés para los y las estudiantes.

Justificación Tecnológica

La tecnología de la programación ha permitido el avance desde niveles bajos a niveles más altos; es decir, lenguajes de programación más parecidas al lenguaje natural. A pesar de estos avances, no existen lenguajes relevantes con palabras reservadas en el idioma español, ni toman en cuenta todo el alfabeto español (la letra ñ y las vocales tildadas). Las posibilidades del desarrollo de este compilador incluso permiten la creación para más lenguajes con palabras reservadas en otros idiomas.

CAPÍTULO 1

Marco Teórico

1.1 - Evolución de los lenguajes de programación

Miles de lenguajes de programación¹ existen, con múltiples dialectos. Algunos tienen un gran y extendido alcance alrededor del mundo, otros se utilizan en entornos muy específicos y los demás han quedado atrás en la historia. Los primeros lenguajes existieron aun antes que las primeras computadoras modernas. La máquina de telar de Joseph Marie Jacquard se controla mediante tarjetas perforadas. Este invento del año 1801 (más de un siglo antes de los ordenadores modernos), a veces considerado la primera computadora, aceptaba “comandos” y patrones a través de las tarjetas. Esto permitía que el usuario de la máquina de telar pueda programar los diseños deseados. Aunque no se considera una computadora, otro ejemplo de dispositivos que responden a la lectura de tarjetas perforadas incluye la pianola, que tuvo uso durante finales del siglo XIX e inicios del siglo XX (Chouchanian, 2010).

Más de tres décadas después de la creación de la máquina de telar de Jacquard, Charles Babbage presentó el concepto de la Máquina Analítica, que permitiría realizar una variedad de cálculos mediante tarjetas perforadas. Esta máquina no fue construida durante la vida de Babbage, pero tuvo un impacto sobre futuros desarrollos de las computadoras. Ada Byron, también conocida como Ada de Lovelace, desarrolló un algoritmo² para la Máquina Analítica, que permitía calcular números de Bernoulli. El desarrollo de este algoritmo representa el primer programa significativo y complejo.

Durante la década de 1930, se vio el desarrollo de los conceptos de la Máquina de Turing y Cálculo Lambda. Aunque estos conceptos no fueron concretados de una manera práctica, estas ideas aportaron considerablemente en el mundo de las ciencias de la computación (Steinberg, 2012).

1.1.1 – Programación a bajo nivel

Las primeras computadoras modernas son las electrónicas que surgieron en la década de los cuarenta. No utilizaban teclados ni los “*mouse*” como conocemos actualmente. Solo se podía utilizar tarjetas perforadas o interruptores para comunicarse con la máquina. Comenzaron con la intención de resolver operaciones matemáticas y algoritmos

¹ **Lenguaje de programación** – Un conjunto de caracteres y reglas sintácticas y semánticas que son entendidas por un compilador o intérprete.

² **Algoritmo** – Una serie de instrucciones lógicas que pretenden resolver un problema de una forma secuencial.

relativamente sencillos (Ferguson, 2004). Aunque eran revolucionarias en su tiempo, hoy se consideran primitivas y obsoletas.

En paralelo con sus capacidades y características va la programación; cada instrucción³ y cada dato⁴ se programaba mediante una secuencia de ceros y unos, los cuales decían directamente al hardware⁵ qué hacer y cómo lograrlo. Esta programación⁶ se conoce como Lenguaje de Máquina. Estas operaciones de bajo nivel⁷ controlan cada aspecto sobre el flujo y los cálculos de cada dato; es decir, la operación realizada sobre ello, su ubicación exacta en memoria y su movimiento por los registros. Era muy difícil programar de esta manera, porque era necesario saber o consultar cada código de operación (*opcode*) y tener versatilidad en los sistemas numéricos binario, octal y hexadecimal. Este proceso era muy lento, y se podía tardar días en desarrollar y ejecutar pequeños algoritmos, y el entendimiento y modificación por terceros no era muy factible. Se considera tan ilegible, que la Copyright Office (Oficina de Derechos de Autor) de los Estados Unidos no podía identificar si un programa particular era una obra original o no. Otra desventaja incluye el hecho de que no se podía correr el código en otra computadora diferente, debido a que cada arquitectura tenía un conjunto de instrucciones diferentes. Un programa de Hola Mundo, convertido a hexadecimal, se podría escribir así: B4 09 8D 16 0D 01 CD 21 B8 00 4C CD 21 48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21 24 (Steinberg, 2012).

1.1.2 – Programación a nivel medio

A finales de la década de los cuarenta, llegaron los primeros lenguajes de nivel medio; lenguaje de ensamblaje. Ensamblaje usa palabras mnemotécnicas para representar a las diferentes instrucciones que un procesador pueda aceptar. Como programar en binario, la programación en ensamblaje permitía realizar funciones de bajo nivel, pero en ensamblaje era más fácil recordar las instrucciones y se podía gestionar las variables⁸ con nombres, en vez de tener que recordar la ubicación exacta en memoria. Ensamblaje también hace el trabajo de manejar saltos de línea de código en bucles y condiciones. Tras escribir código en ensamblaje, se traducía mediante un ensamblador a código binario. Código ensamblaje era

³ **Instrucción** – En la programación, una instrucción dice específicamente lo que el procesador de una computadora debe hacer. Las instrucciones permitidas son aquellas que el procesador puede ejecutar.

⁴ **Dato** – Un valor generalmente numérico o textual que puede ser ingresado, procesado y almacenado, y puede tener salida.

⁵ **Hardware** – Las piezas físicas de una computadora que le dan funcionamiento.

⁶ **Programación** – Es el proceso de llevar un algoritmo, mediante las reglas de un lenguaje de programación, a una serie de caracteres que representan instrucciones y datos.

⁷ **Nivel de Abstracción de un Lenguaje de Programación** – Hace referencia a la cercanía del código a lo que una computadora puede entender.

⁸ **Variable** – Un espacio en memoria que almacena un dato, para usar luego en un cálculo, mostrar al usuario, o almacenar a largo plazo.

mucho más fácil modificar y entender que binario, pero tampoco era portable; como binario, el código era específico para cada arquitectura. Se usa hasta tiempos actuales en electrodomésticos y en controladores de dispositivos (Steinberg, 2012). Un simple programa de Hola Mundo en ensamblaje, para el procesador 6502 versión para Commodore 64, se escribía de esta forma:

```
; Hello World for 6502 Assembler (C64)
ldy #0
beq in
loop:
jsr $ffd2
iny
in:
lda hello,y
bne loop
rts
hello: .tx "Hello World!"
.by 13,10,0
```

1.1.3 – Programación a alto nivel

En el año 1948, Plankalkül, o Plan Cálculo, fue el primer lenguaje de alto nivel desarrollado por Konrad Zuse. Debido a su aislamiento durante la segunda guerra mundial y la destrucción de su trabajo, este lenguaje nunca fue implementada, y pasaron alrededor de nueve años hasta el surgimiento del siguiente lenguaje de programación de alto nivel de gran uso. Aunque se considera de alto nivel, actualmente se ve muy difícil de leer. Su código se escribía de esta forma:

```
R1.1(V0[:sig]) => R0
R1.2(V0[:m x sig]) => R0
0 => i | m + 1 => j
[W [ i < j -> [ R1.1(V0[j: m x sig]) => R0 | i + 1 => i ] ] ] END
R1.3() => R0
'H';'e';'l';'l';'o';';';';';'w';'o';'r';'l';'d';'!' => Z0[: m x sig] R1.2(Z0) => R0
END
```

Empezando a finales de la década de los cincuenta, y durante los años sesenta, aparecieron nuevos lenguajes de alto nivel, como FORTRAN, LISP, ALGOL, COBOL. Estos lenguajes tenían una notación más natural, y mediante un compilador⁹ se traducían a ensamblaje. Esto aumentó considerablemente la portabilidad de código, si existía un compilador para la computadora destina. Estos lenguajes permitían que los programadores escriban código rápidamente sin la necesidad de detenerse a pensar en las características específicas de cada procesador, con la única desventaja de que los compiladores no siempre crean código ensamblaje tan eficiente que hacerlo manualmente. A pesar de la edad que tienen estos lenguajes, la mayoría siguen en uso hasta la actualidad (Ceruzzi & Wexelblat, 1984).

⁹ **Compilador** – Un programa que recibe como entrada un flujo de caracteres de un código fuente y lo transforma en un lenguaje que entiende la computadora (código de máquina o ensamblaje).

FORTRAN nació en el año 1957, creado por John Backus en IBM. Ha tenido muchas revisiones, y sigue en uso. En 1958, LISP y ALGOL fueron diseñados. LISP, creado por John McCarthy en MIT¹⁰, fue el primer lenguaje con una sintaxis descrita usando una notación formal, BNF¹¹ (Forma Backus-Naur), lo cual se ha usado en la gran mayoría de lenguajes posteriores (Ceruzzi & Wexelblat, 1984; Ferguson, 2004).

```
C Fortran: Hello, world! PROGRAM HALLO WRITE (*,100) STOP
100 FORMAT ('Hello, world!') END
```

ALGOL, aunque no se usa hoy en día, inspiró la producción de lenguajes como Pascal, C y Java.

```
BEGIN
DISPLAY ("Hello, World!");
END.
```

En 1959, COBOL nació, siendo inspirado en FLOW-MATIC de Grace Hopper. Su notación fue orientado a usuarios de negocios, por ende, su léxico¹² se asemeja mucho a lenguaje natural (Ceruzzi & Wexelblat, 1984).

```
program-id. hello.
procedure division.
display "Hello, World!".
stop run.
```

Hasta ese tiempo, la programación era algo complejo de aprender, y solo entusiasmados en el campo dedicaron el tiempo de aprenderlo. Viendo la necesidad de un lenguaje más sencillo para los novatos, John Kennedy y Thomas Kurtz desarrollaron BASIC en el año 1964 (Ceruzzi & Wexelblat, 1984). Fue muy popular durante los primeros años de las computadoras personales, ya que permitía que pequeños negocios y usuarios puedan desarrollar y vender sus propias aplicaciones personalizadas. Esto se vio en las computadoras de 8-bit que tuvieron auge dentro de los hogares, que permitían realizar tareas escolares, mantener datos sobre cuentas financieras, tener comunicación con el exterior, e incluso jugar video juegos, en la comodidad del domicilio. Microsoft desarrolló un intérprete¹³ de BASIC para el procesador MOS 6502, que fue usado en varias computadoras personales como los Atari 8-bit, Apple II, Commodore 64, etcétera. Cuando no había ningún periférico de almacenamiento conectado al momento de arranque, algunos de estos ordenadores personales presentaban inmediatamente el intérprete de BASIC. Este lenguaje fue tan popular y sencillo que el código de muchos programas fue compartido mediante revistas, y

¹⁰ MIT – Massachusetts Institute of Technology

¹¹ BNF - Forma Backus-Naur es una notación que representa las gramáticas libres de contexto (cada regla tiene un no terminal que se compone de varios terminales o no terminales). Nunca se incluye dos componentes en el lado izquierdo de la regla. BNF se usa para describir las reglas gramaticales de un lenguaje.

¹² Léxico – El conjunto de palabras aceptadas en un lenguaje.

¹³ Intérprete – Analiza y ejecuta línea por línea las instrucciones de un código escrito, cada vez que se lo ejecuta.

niños podían teclear y ejecutar los programas listados. Un programa Hola Mundo en BASIC se ve así:

```
10 REM "Hello World"
20 PRINT "Hello, World!"
30 END
```

En el año 1966, apareció un nuevo lenguaje: BCPL. Aunque este lenguaje, en sí, no tuvo mucha popularidad, una versión posterior de él, B, fue un predecesor de C.

Hasta ese momento, no existía bien el concepto de la programación estructurada¹⁴; es decir que se usaba con frecuencia comandos como GOTO y JUMP, los cuales hacían más difícil entender el código de un programador. Para la década de los años setenta, se empezó a debatir sobre los méritos de la programación estructurada y cómo facilitaría el entendimiento y seguimiento de código. Los lenguajes que surgieron durante esta década y las futuras reflejan los argumentos presentados a favor de la programación estructurada.

Niklaus Wirth diseñó un nuevo lenguaje en el año 1971, Pascal, con una estructura sencilla en base a bloques, lo cual afectaría el desarrollo de futuros lenguajes.

```
program helloworld;
(* Pascal: Hello, world! *) begin writeln(Hello, World!);
end.
```

En el año 1972, se presentaron grandes avances revolucionarios en el mundo de la programación: la creación de Smalltalk y C. Smalltalk, inventado por Xerox PARC, se considera el primer lenguaje de Programación Orientada a Objetos¹⁵, un paradigma que llegó a tener mucha fama. Su código es de este formato:

```
"Hello World in Smalltalk (simple version)"
Transcript show: 'Hello, World!'.
```

También en ese año, surgió el lenguaje C, creado por Dennis Richie de Bell Laboratories. Es de bajo nivel (entre lenguajes de alto nivel). Es muy rápido y potente, y uno de los lenguajes más usados a nivel mundial, ya que hay pocas arquitecturas computacionales que no tienen compilador C. Con la creación del lenguaje C, fue posible llevar a cabo cualquier tarea de programación, como crear sistemas operativos, compiladores, y otros programas (Steinberg, 2012). Este lenguaje potente se usa hasta el día de hoy en muchos ámbitos.

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
printf("Hello, World!\n"); return EXIT_SUCCESS; }
```

¹⁴ **Programación Estructurada** – Un paradigma de programación que tiene una estructura y secuencia lógicas, sin saltos inesperados. No utiliza comandos como GOTO y JUMP. Utiliza bucles (como while, do-while y for) para evitar el uso de saltos.

¹⁵ **Programación Orientada a Objetos (POO)** – Un paradigma de programación que da la posibilidad de crear objetos con características (atributos) y comportamiento (métodos).

Durante la década de los años ochenta, se decidió que desarrollar procesadores para tener mayores funcionalidades a bajo nivel para los programadores de ensamblaje no era óptimo. Con esta idea en mente, decidieron crear otro tipo de procesador, denominado RISC (*Reduced Instruction Set Computer*, o Computador con conjunto de instrucciones reducido), cuyo desarrollo implicaba mayor facilidad y rendimiento al momento de compilación desde lenguajes de alto nivel (Aho, Lam, Sethi, & Ullman, 2007).

Algunos años después, en el año 1985, se vio el nacimiento de C++, un lenguaje basado en C, pero con Programación Orientada a Objetos, permitiendo un desarrollo más dinámico. C y C++ son muy portables; corren en cualquier sistema que tenía un compilador C o C++, y son fáciles de entender y modificar comparado con código de niveles menores (Steinberg, 2012). A estas alturas, ya no se preocupa por ningún detalle de bajo nivel; muchas de las veces, una línea de código en lenguajes altos puede implicar varias instrucciones para los niveles más bajos.

```
#include <iostream>
int main () {
std::cout << "Hello, World!" << std::endl;
return std::cout.bad();
}
```

En los años noventa, con el advenimiento del Internet, surgieron diferentes lenguajes con usos más específicos, por ejemplo, lenguajes de secuencia de comandos como HTML y JavaScript, y otro lenguaje importante en el ámbito de POO, Java. Java es generalmente entre los primeros lenguajes que se enseñan a los estudiantes de programación. (Chouchanian, 2010) Código JavaScript es así:

```
<script type="text/javascript"> document.write('Hello, world!');
</script>
```

Java tiene este formato:

```
// Java: "Hello, world!"
class HelloWorld {
public static void main(String args[]) { System.out.println("Hello, world!"); }
}
```

C# nació en 2001, siendo un lenguaje multiparadigma, POO de uso general. Aunque nació siendo muy parecido a Java, desde C# 2.0 en 2005, los dos lenguajes han ido divergiendo, con enfoques muy distintos. C# también es un lenguaje muy popular y práctico para comenzar a programar. Un programa Hola Mundo se ve así:

```
System.Console.WriteLine("Hello, World!");
```

Google creó dos lenguajes importantes en los años 2007 y 2009 respectivamente; Go y Dart. Go es un lenguaje de código abierto¹⁶ parecido a C con compiladores para varias

¹⁶ **Código Abierto** – Programas y sistemas de código abierto tienen su código fuente disponible para todos. Se puede realizar cambios y modificaciones para mejorar el software, o agregar funcionalidades.

plataformas. Dart fue creado para reemplazar a JavaScript. Tiene Programación Orientada a Objetos y se puede usar en una amplia variedad de escenarios (Steinberg, 2012).

Go:

```
package main
func main() { println("Hello, World!") }
```

Dart:

```
main() {
var bye = Hello, World!; print("$bye");
}
```

1.1.4 - ¿Por qué existen muchos lenguajes de programación?

El motivo de seguir desarrollando varios lenguajes de programación es para facilitar el trabajo del programador; en la mayoría de los casos, un programador ya no necesita fijarse en ninguna función de bajo nivel, como el movimiento de datos por los registros, la activación o desactivación de las banderas de estado del procesador, cálculos hexadecimales, o saber la ubicación exacta de memoria de un dato o una instrucción. Estas mejoras permiten una programación mucho más fluida, más rápida, menos propensa a errores y más entendible por los colegas.

Como se ha visto, con la evolución de los lenguajes de programación, se ha ido tratando de asemejar estos lenguajes formales a lenguajes naturales, y cada generación trae lenguajes más entendibles. Cada década trae nuevas innovaciones y más facilidades al momento de programar. Viendo la alta cantidad de lenguajes en inglés, se puede suponer que la existencia de lenguajes de programación en otros idiomas podría disminuir el tiempo tomado en aprenderlo para personas que naturalmente hablen otro idioma, como español o francés como su lengua materna.

Basándose en los lenguajes anteriormente descritos, se ha decidido tomar los aspectos que parecen más sencillos para poder armar un nuevo lenguaje de programación que sea relativamente fácil de aprender, cumpliéndose el objetivo del proyecto. Por ello, se va a desarrollar un lenguaje de procedimientos, que use palabras reservadas en español, y tenga una sintaxis relativamente fácil de asimilar. No se va a emplear el uso de llaves, sino dos puntos y "fin" para delimitar el inicio y final de bloques. Se usarán corchetes para referenciar a los arreglos. Se usarán abreviaturas para los tipos de datos para reducir la cantidad de letras tecleadas en el desarrollo de un programa. Aparte de los bucles generales que existen, también habrá otro que acepte un número entero en su declaración, o una expresión que dé un resultado de ese tipo, y se ejecutará las sentencias declaradas en el bucle, sin necesidad de definir condiciones ni variables. El código principal (generalmente contenido dentro de un método *main*) se desarrollará sin requerir la explicitación de tipo de retorno, nombre, parámetros, comienzo, ni fin.

1.2 – Compiladores y sus Fases

1.2.1 - Compiladores: Conceptos Básicos

Un compilador es un programa que traduce un código fuente escrito en un lenguaje de alto nivel a un lenguaje de menor abstracción (código máquina, ensamblaje u otro lenguaje de alto nivel). Una computadora no puede entender código fuente de lenguajes de alto nivel, y se requiere este proceso de compilación para que el procesador pueda recibir instrucciones bien formadas en código de máquina. Existen casos donde el proceso completo de compilación requiera varias compilaciones para llegar al lenguaje destino requerido. Otros términos que se pueden emplear para describir el compilador incluyen conversor, traductor y transformador.

Los lenguajes de programación existen para representar una serie de instrucciones y cálculos que deba realizar un computador, pero se escriben de una forma entendible para los humanos. Mientras mayor sea el nivel de abstracción de un lenguaje, más parecido será a un lenguaje natural; por ende, se lo podrá entender mejor. La desventaja es que generalmente los lenguajes de mayor nivel requieren mayor tiempo, o mayor cantidad de pasos, de traducción o compilación. Además, mientras el lenguaje tenga menos que ver con las funcionalidades internas del hardware, menos control tendrá sobre la ejecución interna del código. Estas desventajas se minimizan tomando en cuenta el hecho de que los lenguajes de alto nivel permiten una construcción rápida de programas, y su fácil entendimiento concede una depuración más viable.

Los compiladores se encargan de llevar esta notación hacia un formato ejecutable con el mismo significado, o semántica, que el código fuente, de forma que el procesador la pueda entender. La entrada es una cadena de caracteres, o un texto de código fuente. En el caso de compiladores de varias pasadas, alguna representación del código será extraído como un resultado en cada paso, y será el insumo para el siguiente. Los compiladores no son diferentes que otros programas en el hecho de que aceptan una entrada, procesan los datos, y producen una salida como resultado.

Un compilador se diferencia de un intérprete en un aspecto fundamental; el compilador traduce el código mediante una serie de pasos a un archivo ejecutable; esto se hace una vez. En cambio, un intérprete debe interpretar y ejecutar cada línea de código a la vez, en cada corrida. Este proceso de interpretación ocurre cada vez que se ejecuta el programa. Debido a esta diferencia, generalmente hay un rendimiento mayor en programas compilados sobre los interpretados (Aho, Lam, Sethi & Ullman 2007). Actualmente, incluso los intérpretes realizan algunas fases de compilación antes de interpretar el código (Brookshear, 1993).

Existen diferentes tipos de compiladores, como los ensambladores, meta-compiladores, compiladores cruzados, compiladores de varias pasadas, de una pasada, etcétera, cada una con su respectiva función y uso (Brookshear 1993; ITS Education Solutions Limited 2012; Maurya 2011). Se va a enfocar específicamente en los compiladores de una sola pasada.

Una compilación se puede realizar de fuente a binario (alto nivel a bajo nivel), de fuente a ensamblaje (alto nivel a nivel medio), o de fuente a fuente (alto nivel a alto nivel). Pueden ocurrir algunas compilaciones para llegar al código ejecutable.

Los compiladores tienen la tarea no solo de traducir código, sino de avisar sobre errores presentes, sugerir cambios, y asegurar que el código destino cumpla la misma funcionalidad descrita en el código original. Existen dos conjuntos de pasos: el primero se denomina el front-end, o el análisis, mientras que el segundo es el back-end; la síntesis. El análisis averigua la correcta escritura léxica, divide la entrada en tokens, verifica una gramática adecuada y una escritura semántica apropiada. Después de culminar estos procesos, genera un código intermedio. La síntesis es la construcción del código destino a partir del código intermedio (Aho et al., 2007).

El análisis tiene: analizador léxico (también denominado análisis lineal), analizador sintáctico (o jerárquico) y analizador semántico (o de restricciones contextuales), más un generador de código intermedio. Cada paso requiere de una tabla de símbolos (TDS), y ésta es modificada y accedida por cada analizador y por el generador de código intermedio. Luego, en la síntesis, existe como mínimo el generador de código destino, aunque pueda haber una u más optimizadores de código. Tal como el análisis, la síntesis requiere de la tabla de símbolos (Aho et al., 2007). Aparte de la tabla de símbolos, se puede incluir el manejo de otras tablas o descriptores para el almacenamiento de bucles, tiras de caracteres o constantes (Brookshear, 1993).

1.2.2 - Análisis Léxico

La primera fase de compilación comienza con el analizador léxico, también llamado el scanner o explorador. Su entrada es un flujo de caracteres, que es el código fuente a compilar. Al recibir el texto fuente, el primer paso es realizar una “limpieza” previa de los caracteres. Esto incluye eliminar espacios, líneas vacías y también comentarios. Este primer paso es opcional, pero ayuda en la eficiencia de la segunda parte del scanner (Aho et al., 2007). El siguiente paso es convertir el código fuente en un conjunto de tokens¹⁷ (componentes léxicos) que serán la entrada de la siguiente fase, el analizador sintáctico.

¹⁷ **Token** – Es una representación de un lexema y consiste en un nombre y un atributo opcional.

Para realizar la transición de caracteres a tokens, es necesario comprobar que cada palabra pertenezca al lenguaje. Las palabras son tiras de caracteres que pertenecen al léxico de un lenguaje. Cuando alguien escribe una oración en lenguaje natural, debe asegurar que escriba las palabras de una forma ortográficamente precisa, asegurando que cada vocablo pertenezca al lenguaje. En el idioma español, se puede redactar la oración “El gato tomma leche.” En este ejemplo, las tres tiras “El”, “gato” y “leche” corresponden a palabras de la lengua española. En el caso de “tomma”, contiene una ‘m’ adicional, que la excluye del idioma. El analizador léxico realiza esta función: determina que todos los lexemas¹⁸ escritos se hallen dentro del lenguaje.

La definición léxica del lenguaje se lo puede hacer de dos formas comunes: mediante expresiones regulares (ER)¹⁹ o empleando autómatas finitos (AF)²⁰. Expresiones regulares son una notación que permite describir las reglas que deban tener las palabras que pertenecen a un lenguaje. Mediante los operadores de concatenación, unión y la clausura de Kleene²¹, y los caracteres del alfabeto²² del lenguaje, se puede especificar la escritura de todas sus palabras. ‘*’ representa la clausura de Kleene, lo que significa que una tira de caracteres se puede repetir hasta una cantidad infinita de veces, o no estar presente. Otra manera de referirse a este operador es “cero o más instancias”. Basándose en este operador, se puede también tomar en cuenta los operadores de la clausura positiva de Kleene ‘+’, que significa una o más instancias, y ‘?’, de cero o una instancia. Otra facilidad que ha surgido en los últimos años incluye las clases de caracteres, donde se puede agrupar varios símbolos que tengan secuencia lógica mediante una abreviación (Aho et al., 2007).

Antes de ejemplificar estos conceptos, es necesario saber que los operadores ‘*’, ‘+’ y ‘?’ tienen máxima precedencia en el orden de operaciones. Concatenación, que se puede representar con dos caracteres juntos, tiene precedencia media. Unión, representado por ‘|’, tiene la menor precedencia (Aho et al., 2007).

En el caso de un alfabeto muy sencillo, que consista del conjunto {a,b,c,0,1,2,3,+,-}, se puede crear reglas, como:

```
letra → a | b | c
dígito → 0 | 1 | 2 | 3
signo → + | -
```

¹⁸ **Lexema** – Una secuencia de caracteres del código fuente que acople a un patrón. Es una unidad léxica.

¹⁹ **Expresión Regular** – Es una notación o patrón que describe qué caracteres y en qué orden producen un lexema específico.

²⁰ **Autómata Finito** – Se usa para definir el léxico de un lenguaje mediante gráficos. Tiene un conjunto finito de estados y transiciones.

²¹ **Clausura de Kleene** – Simbolizado con un asterisco, este operador se utiliza para especificar que una tira se puede repetir cualquier cantidad de veces o no estar presente.

²² **Alfabeto** – El conjunto de caracteres (letras, números y símbolos) que un lenguaje acepte.

Tomando en cuenta las clases de caracteres, se podría resumir letra y dígito de la siguiente manera:

letra \rightarrow [abc]
dígito \rightarrow [0123]

Un identificador que empieza con una letra seguida por cero o más instancias de letras y dígitos tendría la siguiente regla:

id \rightarrow letra (letra | dígito)*

Si se desea representar un número entero, que incluya al menos un dígito, se puede hacer con esta notación:

número \rightarrow dígito+

Para expandir esta regla, se puede incluir un signo positivo o negativo, o incluso omitirlo: número se compone de cero o una instancia de signos seguido por una o más instancias de dígitos.

número \rightarrow signo? dígito+

Mediante esta notación de expresiones regulares, se puede definir con facilidad las palabras que un lenguaje pueda reconocer; a esta la denominamos “formal.”

Los autómatas finitos (AF) son otra manera de definir el léxico de un lenguaje formal. Son gráficos que incluyen un conjunto de estados y transiciones entre estados, un estado inicial, y un conjunto de estados de aceptación. Existen dos tipos de autómatas finitos, que son los deterministas y los no deterministas. Los autómatas finitos no deterministas, o AFN, pueden tener transiciones ambiguas; es decir, múltiples transiciones desde un estado con el mismo carácter. Además, existen transiciones con ‘ ϵ ’, la cadena vacía. Estas dos condiciones de los AFN crean mucha ambigüedad. Aunque es más fácil diseñar un autómata no determinista, es más difícil implementar que los autómatas finitos deterministas, o AFD. Los AFD tienen máximo una transición por carácter en un estado dado, y no existe ninguna transición con la cadena vacía (Aho et al., 2007). Los AFD se representan por $(S, \Sigma, \delta, i, F)$, donde S es un conjunto finito de estados, Σ representa el alfabeto, δ significa la función de transición, i es el estado inicial (que es un elemento de S) y F el conjunto de estados de aceptación (subconjunto de S) (Sanchis Llorca & Galán Pascual 1986).

Es más difícil desarrollar un AFD, pero es más eficiente que un AFN al reconocer palabras. Los AFD y los AFN pueden reconocer los mismos lenguajes (Sanchis Llorca & Galán Pascual 1986).

Es común crear expresiones regulares para explicitar la forma de las palabras del léxico. Basándose en las expresiones regulares, que son más fáciles para la lectura humana, se

suele crear AFN, y a partir de ello se procede a crear un AFD. El diagrama del AFD se puede representar mediante una tabla de transiciones, que incluye los estados como filas, los símbolos de entrada como columnas, y se ingresa los estados de llegada en las celdas. Esto provee un método explícito al analizador léxico para determinar qué lexemas pertenecen al lenguaje, y facilita su transición a tokens (Aho et al., 2007).

Al contemplar palabras reservadas²³, lexemas con una escritura específica y un uso particular, si tienen la forma de identificador, se puede incluir cada palabra con su propia expresión regular e implementarlo en el AFD. Hacer esto aumentará considerablemente la cantidad de estados del autómata, y aumentará los recursos requeridos para crear el autómata y mantener la matriz de transiciones. Otra posibilidad es incluir las palabras reservadas en una tabla especial en el programa, y cuando el analizador lea un identificador, puede consultar primero esa tabla para verificar si es una palabra reservada o si realmente es un identificador. Este método puede resultar en un autómata mucho más pequeño y tiene la ventaja de mayor flexibilidad al momento de aumentar o quitar palabras reservadas (Aho et al., 2007).

Cuando existan errores léxicos, es decir, cuando un símbolo de entrada no tenga un estado de llegada desde el estado actual, se puede producir directamente un error, o emplear una técnica de recuperación. Existen cinco de estos métodos. El modo de pánico elimina símbolos donde está el cursor hasta hallar un lexema aceptado. El siguiente incluye eliminar un carácter que sobre para formar una palabra reconocida. El tercer método es insertar un carácter que falte. Luego hay el método de reemplazar un carácter por otro. Por último, se puede transponer dos caracteres adyacentes. Idealmente, se debe emplear el método de recuperación que requiera la menor cantidad de permutaciones, aunque este cálculo puede ser a veces ineficiente y requerir mucho tiempo y poder computacional. Se debe emplear algunas de estas técnicas, y recurrir a “modo pánico” como la última posibilidad (Aho et al. 2007; Brookshear 1993; ITL Education Solutions Limited 2012; Maurya 2011).

El uso de una estructura de tabla o matriz para almacenar los estados de un autómata es más eficiente en cuestión de cantidad de tiempo tomado en recorrerlo; pero si el autómata es muy grande, con muchos estados y entradas del alfabeto, es muy conveniente usar una tabla compacta. Esta estructura reduce en gran cantidad la memoria requerida por el autómata, aunque requiere mayor cantidad de ciclos de reloj del procesador para realizar la recorrida. Un autómata de cien estados con cien entradas con ciento diez transiciones

²³ **Palabra Reservada** – Un lexema que suele ser un identificador bajo circunstancias normales, pero es reservado para un uso distinto y específico.

ocuparía un total de diez mil unidades de memoria. Una tabla compacta reduciría esta cantidad a meramente 420 unidades (Aho et al., 2007; Brookshear, 1993).

Se suele mantener la fase de análisis léxico separado del analizador sintáctico porque aumenta la facilidad de diseño, mejora la eficiencia del compilador, y hace que el compilador sea más portable. Si fuera necesario que el analizador sintáctico maneje espacios en blanco, comentarios, etcétera, su diseño e implementación serían de mayor dificultad (Aho et al., 2007).

Otra tarea del scanner es agregar valores a la tabla de símbolos, lo cual se utilizará durante cada fase de compilación. La tabla de símbolos es donde se almacenan variables y constantes, su nombre, valor y tipo de dato. Aunque no necesariamente se va a poder adquirir toda esta información durante el análisis inicial, se puede empezar a construirla (Aho et al., 2007).

1.2.3 - Análisis Sintáctico

La siguiente fase de compilación realiza un análisis sintáctico, también llamado el *párser* o reconocedor. El analizador sintáctico verifica que el flujo de tokens que tiene de entrada se acople a las reglas gramaticales del lenguaje. No se puede utilizar los AFD normales ni expresiones regulares para describir las gramáticas, debido a que los AFD simples no tienen memoria. Es necesario que el instrumento usado para verificar una gramática pueda recordar. Por ejemplo, en una gramática con las siguientes reglas:

$$\begin{aligned} E &\rightarrow E + T \\ T &\rightarrow F \\ F &\rightarrow (E) \end{aligned}$$

no se va a poder recordar cuántos paréntesis se haya recorrido mediante un AFD simple. Para este fin, se podría utilizar un “autómata de pila” (AP)²⁴. Estas “máquinas” funcionan igual que un AFD, con la excepción de que tienen una pila, y las transiciones incluyen símbolos para agregar o quitar de ella (Sanchis Llorca & Galán Pascual 1986).

Se puede definir una gramática de la forma $G = (N, T, S, R)$ donde N es un conjunto finito de no terminales, T representa un conjunto finito de terminales, S es el símbolo inicial, lo cual es un elemento de N y R representa un conjunto finito de reglas de reescritura. En el ejemplo presentado anteriormente, $N = \{E, T, F\}$, $T = \{+, (,)\}$, $S = E$ y $R = \{E \rightarrow E + T, T \rightarrow F, F \rightarrow (E)\}$. Usando las reglas de reescritura, se podrá definir una gramática libre de contexto (también llamado BNF) (Sanchis Llorca & Galán Pascual 1986).

²⁴ **Autómata de Pila** – Un autómata finito con un elemento adicional: una pila que almacena símbolos que ayudan a determinar los siguientes movimientos.

Usando las reglas de reescritura de una gramática, y basándose con el símbolo inicial, se puede construir un árbol de derivación para cada instancia de código escrito siguiendo las reglas gramaticales. Los árboles de derivación tienen terminales como hojas, y símbolos no terminales que componen los nodos interiores. Lo ideal es que se pueda construir únicamente un árbol distinto por instancia de código. En los casos donde haya más de un árbol, significa que la gramática es ambigua, y no habrá una opción explícita al momento de derivar cada regla.

$$E \rightarrow E + E \mid E * E \mid id$$

Si se basara en las reglas anteriores y el enunciado `id + id * literal`, se podría construir el árbol de dos formas, como se indica en la figura 3.

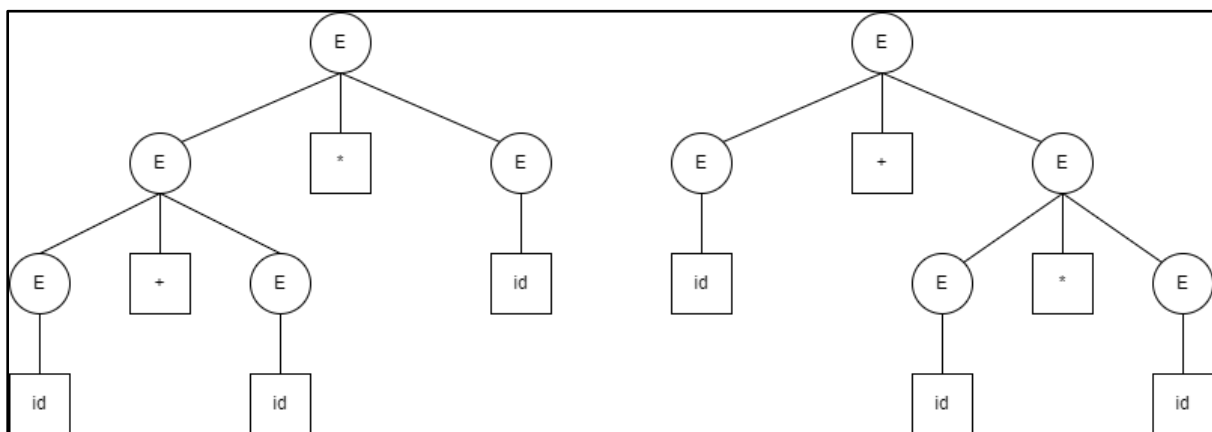


Fig. 3. Árboles de derivación que muestran ambigüedad tras tener dos posibilidades con un solo enunciado: `id + id * id`.

Como se puede observar en la Figura 1, existen dos árboles de derivación distintos para un solo enunciado. Esto genera ambigüedad, lo cual la mayoría de los analizadores sintácticos no pueden tolerar. Para resolver ello, se debe eliminar reglas ambiguas.

Tras el desarrollo general de una gramática, se lo debe simplificar, viendo la vacuidad del lenguaje²⁵, realizando la supresión de símbolos no terminales²⁶ y símbolos inaccesibles²⁷, y eliminando reglas lambda²⁸ y unitarias²⁹. Otra manera de simplificar las gramáticas es mediante el uso de una forma normal, como la forma normal de Chomsky o la de Greibach (Brookshear, 1993).

²⁵ **Vacuidad del Lenguaje** – Una gramática es vacía si es imposible llegar a una tira terminal desde el nodo raíz.

²⁶ **Símbolo no Terminable** – Es un símbolo no terminal de una gramática que no puede llegar a tener un nodo terminal. Se debe eliminar estos símbolos por cuestiones de simplicidad.

²⁷ **Símbolo Inaccesible** – Es un símbolo terminal o no terminal de una gramática, al cual no se puede llegar mediante las reglas gramaticales existentes desde el nodo raíz.

²⁸ **Regla Lambda** – Es una regla gramatical de la forma $A \rightarrow \epsilon$ o $A \rightarrow \lambda$. Tener estas reglas puede complicar o incluso imposibilitar la creación del analizador.

²⁹ **Regla Unitaria** – Las reglas de la forma $A \rightarrow B$ (un símbolo no terminal a otro símbolo no terminal) so unitarias.

Al momento de desarrollar un analizador sintáctico, se debe escoger entre los tres tipos existentes, que son universales, descendentes y ascendentes. Los métodos de análisis universal no son eficientes ni óptimos para uso en un compilador; de preferencia, se debe emplear un analizador descendente o ascendente. Los analizadores descendentes construyen el árbol de derivación y reconocen las reglas empezando desde el nodo raíz, y van construyendo hacia las hojas. En cambio, un ascendente trabaja al revés; empieza desde las hojas y va construyendo el árbol hacia arriba hasta la raíz (ITL Education Solutions Limited 2012).

Los analizadores descendentes son más fáciles de construir y más rápidos al momento de reconocer la cadena de entrada. Sin embargo, tienen algunas desventajas: tienen más restricciones que los analizadores ascendentes en cuanto a la gramática y sus reglas, y en general son menos eficientes. Una de sus restricciones incluye el no poder manejar recursión por la izquierda y la necesidad de eliminar reglas que tengan esta característica. Los ascendentes aceptan una mayor cantidad de gramáticas y pueden tolerar recursión por ambos lados, y pueden detectar errores rápidamente. Desafortunadamente, es mucho más difícil construir este tipo de analizador; para su construcción, conviene usar algún programa ya desarrollado en vez de emplear los algoritmos manualmente (Sanchis Llorca & Galán Pascual 1986).

En décadas pasadas, se usaba con frecuencia *pársers* descendentes de tipo descenso recursivo (Brookshear, 1993). Este analizador que se podía generar fácilmente, sin necesidad de una tabla, simplemente recorría cada regla de la gramática hasta encontrar un árbol de derivación que sea congruente con la entrada de tokens reconocidos. Esta técnica no es eficiente, ya que se puede perder mucho tiempo probando las diferentes reglas hasta encontrar la correcta. En algunos casos, implica regresar hacia arriba en el árbol debido a la mala elección de una regla. Otro método de análisis descendente es el método LL(1)³⁰, que requiere la creación previa de una tabla que ayude en la decisión de qué reglas reconocer durante el análisis de los tokens. Este tipo de analizador no es recursivo, y no tiene las desventajas de rendimiento que el método de descenso recursivo. Para usar el algoritmo de construcción de la tabla LL(1), primero se debe calcular las listas PRIMERO y SIGUIENTE. El algoritmo de PRIMERO(X) se aplica para todos los símbolos gramaticales X hasta no poder añadir más terminales o ϵ :

1. Si X es terminal entonces PRIMERO(X) es {X}
2. Si $X \rightarrow \epsilon$ es una producción, entonces añadir ϵ a PRIMERO(X)

³⁰ **LL(1)** – Left-to-right, leftmost derivation. Reconoce las reglas, con la lectura desde la izquierda y un reconocimiento de reglas por la izquierda. Usa un símbolo de predicción.

- Si X es no terminal y $X \rightarrow Y_1 Y_2 \dots Y_n$, entonces poner a en $\text{PRIMERO}(X)$ si, para alguna i , a está en $\text{PRIMERO}(Y_i)$ y ϵ está en todos los $\text{PRIMERO}(Y_i), \dots, \text{PRIMERO}(Y_{i-1})$; es decir $Y_i \dots Y_{i-1} \Rightarrow \epsilon$.

Luego, se debe calcular la lista $\text{SIGUIENTE}(X)$. Se aplica a los no-terminales, hasta no poder añadir nada más a ningún conjunto SIGUIENTE .

- Póngase $\$$ en $\text{SIGUIENTE}(S)$ donde S es el axioma o símbolo inicial y $\$$ es el delimitador derecho de la entrada.
- Si hay una producción $A \rightarrow \alpha B \beta$, entonces todo lo que está en $\text{PRIMERO}(\beta)$ excepto vacío se pone en $\text{SIGUIENTE}(B)$.
- Si hay una producción $A \rightarrow \alpha B$ o una producción $A \rightarrow \alpha B \beta$, donde $\text{PRIMERO}(\beta)$ contenga ϵ , entonces todo lo que esté en $\text{SIGUIENTE}(A)$ se pone en $\text{SIGUIENTE}(B)$.

Al tener completos los conjuntos PRIMERO y SIGUIENTE , se puede emplear el siguiente algoritmo para construir la tabla $\text{LL}(1)$:

- Para cada producción $A \rightarrow \alpha$ de la gramática, darse los pasos 2 y 3.
- Para cada terminal A de $\text{PRIMERO}(\alpha)$, añádese $A \rightarrow \alpha$ a $M[A, a]$.
- Si ϵ está en $\text{PRIMERO}(\alpha)$, añádese $A \rightarrow \alpha$ a $M[A, b]$ para cada terminal b de $\text{SIGUIENTE}(A)$. Si ϵ está en $\text{PRIMERO}(\alpha)$ y $\$$ está en $\text{SIGUIENTE}(A)$, añádese $A \rightarrow \alpha$ a $M[A, \$]$.
- Hágase que cada entrada no definida de M sea error.

Con la siguiente gramática, se puede calcular los conjuntos PRIMERO y SIGUIENTE , y construir la tabla del analizador, como se muestra en la tabla 1.1:

$E \rightarrow TE' \$$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid id$

$\text{PRIMERO}(E) = \text{PRIMERO}(T) = \text{PRIMERO}(F) = \{ (, id \}$

$\text{PRIMERO}(E') = \{ +, \epsilon \}$

$\text{PRIMERO}(T') = \{ *, \epsilon \}$

$\text{SIGUIENTE}(E) = \text{SIGUIENTE}(E') = \{), \$ \}$

$\text{SIGUIENTE}(T) = \text{SIGUIENTE}(T') = \text{SIGUIENTE}(E') \cup \text{PRIMERO}(E') = \{ +,), \$ \}$

$\text{SIGUIENTE}(F) = \text{SIGUIENTE}(T') \cup \text{PRIMERO}(T') = \{ +, *,), \$ \}$

TABLA 1.1 Tabla de análisis sintáctico $\text{LL}(1)$.

NO TERMINAL	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Existen algunos tipos comunes de los analizadores ascendentes: LR(0)³¹, SLR³², LR(1)³³ y LALR³⁴. LR(1) permite detección instantánea de errores, al costo de tener muchos estados más que los otros tipos de analizadores. Acepta más gramáticas que los otros tipos. SLR tiene menos estados, pero se demora más en detección de errores que LR(1), y permite menos gramáticas. LALR queda entre los dos en términos de gramáticas aceptadas, pero mantiene el tamaño reducido de estados de SLR, y también se puede demorar algunos pasos para la detección de errores. En la mayoría de los casos, es conveniente escoger LALR. Con estos analizadores ascendentes, hay que tener cuidado con indecisiones, que se producen cuando hay conflictos reducir-reducir o desplazar-reducir. Esto se produce cuando el generador del analizador no sabe con exactitud qué decisión tomar en una celda de la tabla del *reconocedor* (Maurya 2011).

La tabla 1.2 muestra un ejemplo de cómo se ve una tabla de análisis sintáctico LR, con los desplazamientos (s), reducciones (r), aceptación (acc) o error (espacios en blanco).

TABLA 1.2 Tabla de análisis sintáctico LR.

Estado	ACCIÓN					IR A			
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Para recorrer los tokens y la tabla de análisis sintáctico con pársers ascendentes, se debe usar el siguiente algoritmo:

³¹ **LR(0)** – Left-to-right, rightmost derivation, canonical. Reconoce las reglas, con la lectura desde la izquierda y un reconocimiento de reglas por la derecha. No usa un símbolo de predicción.

³² **SLR** – Simple LR.

³³ **LR(1)** - Left-to-right, leftmost derivation. Reconoce las reglas, con la lectura desde la izquierda y un reconocimiento de reglas por la derecha. Usa un símbolo de predicción.

³⁴ **LALR** – Look-ahead LR.

```

Hacer que  $a$  sea el primer símbolo de  $w$ ;
while(1) { /* repetir indefinidamente */
    hacer que  $s$  sea el estado en la parte superior de la pila;
    if ( ACCIÓN[ $s,a$ ] = desplazar  $t$ ) {
        meter  $t$  en la pila;
        hacer que  $a$  sea el siguiente símbolo de entrada;
    } else if (ACCIÓN[ $s,a$ ] = reducir  $A \rightarrow \beta$ ) {
        sacar  $|\beta|$  símbolos de la pila;
        hacer que el estado  $t$  ahora esté en la parte superior de la pila;
        meter IR  $A[t,A]$  en la pila;
        enviar de salida la producción  $A \rightarrow \beta$ ;
    } else if (ACCIÓN[ $s,a$ ] = aceptar) break; /* terminó el análisis sintáctico */
    else llamar a la rutina de recuperación de errores;
} (Aho et al., 2007)

```

Estos analizadores, al generar una tabla, tienen cuatro posibles acciones en cada estado con un token dado: reducir la regla, desplazar el siguiente token a la pila sintáctica, aceptar o arrojar un error. Se puede llenar las celdas vacías, las que generan error, con números que lleven a rutinas de recuperación de errores.

Durante esta fase, el analizador sintáctico debe ir consultando y modificando la tabla de símbolos, como se debe de cierto modo, en todas las fases de compilación. Tras reconocer el flujo de tokens de entrada y validar su correcta construcción, se debe pasar el árbol de derivación generado (o en muchos casos, se genera una pila sintáctica) al analizador semántico.

1.2.4 - Análisis Semántico

El analizador semántico recibe el flujo de tokens aceptado por la fase de análisis sintáctico y realiza una serie de operaciones y averiguaciones sobre los tokens. Aunque algunos procesos de esta fase de análisis se pueden realizar durante la generación del árbol sintáctico, es preferible mantener estos dos procesos separados por cuestiones de sencillez y modularidad del código.

Una de las tareas principales de esta etapa es la comprobación de tipos, que verifica el correcto uso de datos, identificadores, y operadores en cuanto a su tipo. De preferencia se debe mantener el mismo tipo de dato entre los operandos de una expresión, pero existen casos donde pueda haber tipos diferentes y aún ser código aceptable; por ejemplo, puede haber las siguientes instrucciones:

```

int i = 5;
double d = i + 42.3 * 2;

```

La segunda instrucción tiene dos tipos de datos diferentes: **double**³⁵ (*d* y 42.3) e **int**³⁶ (*i* y 2). Mediante una conversión implícita de un tipo de dato menor a mayor, en este caso de **int** a **double**, llamada ampliación, se puede validar y generar código intermedio. En la siguiente instrucción, no es posible pasar el paso de comprobación de tipos:

```
int i = 5 + "string";
```

porque el tipo entero no tiene compatibilidad con el tipo cadena de caracteres³⁷, y el operador + implica una suma cuando trata de tipos de datos numéricos, y de concatenación³⁸ cuando hay cadenas de caracteres.

El analizador semántico también debe encargarse de obtener cierta información y pasarla a nodos vecinos del árbol donde sea pertinente. En el siguiente ejemplo, se tomará en consideración las siguientes producciones gramaticales:

$A \rightarrow id = E$

$E \rightarrow E + T$

$T \rightarrow F$

$F \rightarrow \text{dígito}$

Al analizar una tira como $a = 5 + 3$, se podría generar el siguiente árbol de la figura 4:

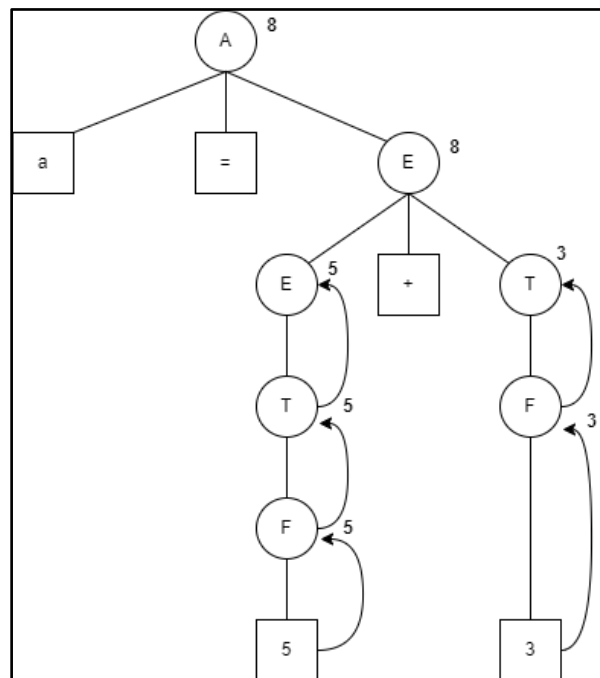


Fig. 4. Árbol de derivación: $a = 5 + 3$.

³⁵ **Double** – Tipo de dato que tiene mucha precisión. En algunos lenguajes de programación, usa 8 bytes de memoria.

³⁶ **Int** – Tipo de dato numérico entero. Usa 4 bytes de memoria.

³⁷ **Cadena de caracteres (string)** – Es un tipo de dato que permite el ingreso de varios caracteres alfanuméricos y símbolos.

³⁸ **Concatenación** – La unión de dos datos de tipo string, para formar uno solo.

El valor léxico de los nodos terminales (con excepción de + y =, que no requieren de atributos adicionales) sería su mismo valor; es decir, 'a', '5' y '3' respectivamente. El valor numérico asignado a F, padre del terminal 5, será el valor sintetizado de su hijo, 5. T y luego E recibirán este mismo valor. En el lado derecho, F y luego T reciben el valor 3. El primer E en el árbol, debe evaluar la operación de la suma de los valores de sus hijos E y T, asignándole el valor de 8. Este 8, al final, se heredará a A. Los nodos del árbol pueden tener atributos de dos tipos: atributos sintetizados y heredados. Los atributos sintetizados se calculan únicamente de los atributos de los nodos hijos, y los heredados se pueden computar desde nodos hermanos o nodos padres (Aho et al., 2007).

Las definiciones dirigidas por la sintaxis especifican los valores de atributos asociando reglas semánticas con las producciones gramaticales, como $E \rightarrow E_1 + T \rightarrow E_{code} = E_{1.code} || T_{code} || '+'$. También existen los esquemas de traducción orientados por la sintaxis, que son gramáticas libres de contexto que contienen acciones semánticas dentro de sus producciones, como $E \rightarrow E_1 + T \{print '+'\}$. La ventaja de las definiciones dirigidas por la sintaxis es que son más legibles que los esquemas de traducción, pero a costo de no ser tan eficientes. Aunque este paso se puede realizar tras la generación del árbol sintáctico, también se puede realizar durante la fase del analizador sintáctico, sin un árbol explícito (Aho et al., 2007).

Existen dos tipos de traducciones, que son de atributos sintetizados, y de atributos heredados por la izquierda. Los esquemas de traducción de atributos sintetizados son fáciles de implementar en *pársers* ascendentes, y contienen solamente atributos sintetizados. Por el otro lado, los esquemas de atributos heredados por la izquierda son un poco más difíciles de implementar, pero pueden contener atributos sintetizados y heredados, y se pueden realizar con cualquier tipo de *párser* (Aho et al., 2007).

Los gráficos de dependencias se utilizan para representar el orden de cálculo de atributos del árbol, el flujo de información entre los nodos y la dirección que toma. Esto ayuda también a prevenir las definiciones circulares, las cuales pueden causar inconvenientes al haber ambigüedad en el camino a elegir al evaluar reglas semánticas. (Aho et al., 2007).

Al culminar la fase de análisis semántico, se debe entregar el árbol sintáctico, modificado y ahora llamado el árbol semántico³⁹, a la siguiente fase de compilación, que es la generación de código intermedio. El nuevo árbol tendrá todos los atributos requeridos en

³⁹ Como se mencionó anteriormente, no siempre es necesario usar un árbol sintáctico; se puede manejar una pila sintáctica que posteriormente se llamará la pila semántica.

cada nodo del árbol sintáctico anterior, y verificado el uso correcto de los tipos de datos en él.

1.2.5 - Generación de Código Intermedio

La siguiente fase de compilación es el generador de código intermedio. El código intermedio generado debe encontrarse en un nivel de abstracción entre el código fuente y el lenguaje destino. Esto tiene varias ventajas. Entre ellas, se podría mencionar que el código intermedio es independiente de la máquina, lo que facilita modificar solo la última fase, generación de código objeto, para obtener código destino para otro tipo de procesador. También permite optimizaciones independientes de la máquina (ITL Education Solutions Limited 2012).

El código intermedio consta de instrucciones de tres direcciones (operador, operando 1, operando 2). La mayoría de las instrucciones constan de un resultado, dos operandos y un operador. Por ejemplo, $x = y + z$. En este ejemplo de asignación, tenemos x como el resultado, y y z como operandos, $+$ como operador. Existen algunas excepciones a la regla, en los siguientes casos especiales:

- Asignaciones, que toman la forma $x = op\ y$
- Instrucciones de copia: $x = y$
- Saltos incondicionales que toman la forma `goto L`
- Saltos condicionales que sean `if x goto L` o `ifFalse x goto L`
- Saltos condicionales parecidos a `if x relop y goto L`, donde `relop` es un operador relacional
- Llamadas a procedimientos, y retornar de ellos:
 - parámetro x_1 (hasta x_n)
 - llamar p,n
- Instrucciones de copia indexadas: $x = y[i]$ o $x[i] = y$
- Asignaciones de direcciones y punteros $x = \&y$, $x = *y$ y $*x = y$

(Aho et al., 2007).

Uniendo estas formas del código de tres direcciones, se puede armar cualquier programa, suponiendo que las instrucciones implicadas no superen el poder computacional e instruccional del procesador en cuestión.

Existen diferentes formas de implementar código de tres direcciones; cuádruplos, tripletas y tripletas indirectas. Los cuádruplos tienen el formato mencionado anteriormente: (operador, argumento 1, argumento 2, resultado) con las excepciones ya detalladas. El ejemplo de la

tabla 1.3 muestra dos asignaciones, o instrucciones de copia, y una suma (Aho et al., 2007; Brookshear, 1993).

TABLA 1.3 Representación de código de tres direcciones mediante cuádruplos: $a = 5$, $b = 2$, $c = a + b$.

#	Op	Arg1	Arg2	Res
0	=	5		a
1	=	2		b
2	+	a	b	c

Las tripletas son parecidas, pero no tienen resultado; resultados temporales se referencian mediante un código de cada línea. Por ejemplo, se podría tener el código anterior en forma de tripletas como se indica en la tabla 1.4:

TABLA 1.4 Representación de código de tres direcciones mediante tripletas: $a = 5$, $b = 2$, $c = a + b$.

#	Op	Arg1	Arg2
0	=	5	
1	=	2	
2	+	[0]	[1]

En este ejemplo, los argumentos para la línea #2 hacen referencia a los valores almacenados temporalmente en las líneas #0 y #1, respectivamente.

Las tripletas indirectas son muy parecidas a las tripletas, con la excepción de que se agrega otra tabla que contiene punteros a las tripletas. Esto permite un reordenamiento durante la fase de optimización que no afectará las tripletas, ni requerirá mucho poder de procesamiento para cambiar. Esta representación permite un ahorro considerable de espacio cuando aparecen muchos tercetos idénticos (Aho et al., 2007; Brookshear, 1993).

Una última forma de describir código intermedio es mediante la forma de asignación individual estática, lo cual usa variables con nombres distintos una única vez (Aho et al., 2007). Por ejemplo, considerando las siguientes instrucciones:

```
a = 5
b = a * 2
a = a + b
```

En la forma de asignación individual estática, se vería así:

```
a1 = 5
b1 = a1 * 2
a2 = a1 + b1
```

Existen otros métodos de representación intermedia, como la Notación Polaca Inversa (RPN), árboles sintácticos abstractos y grafos acíclicos dirigidos, con una dificultad que

incrementa de forma respectiva pero también implica mayor control sobre el producto final (Brookshear, 1993; Maurya, 2011).

Aparte de la generación de código de tres direcciones, la fase de generación de código puede realizar otras funciones también, como la verificación (comprobación) de tipos, cálculo de espacio estático requerido, entre otros. Al momento de ejecutar un programa, existen dos espacios principales de memoria que se usarán: el espacio de las instrucciones, o el código, y el espacio de los datos. El tamaño del espacio requerido para almacenar las instrucciones se puede calcular durante la compilación, pero no siempre se sabe el tamaño exacto de los datos. El espacio requerido por datos de tipos sencillos (también denominados “primitivos”) declarados en el código se puede computar durante compilación, pero otros tipos de datos más complejos, como arreglos, no siempre se sabrá su tamaño hasta el tiempo de ejecución (Aho et al., 2007).

El espacio de los datos se puede dividir en cuatro partes diferentes: datos estáticos, montículo, memoria libre y la pila. Los **datos estáticos** son aquellos datos sencillos declarados en el código cuyo espacio es calculable durante compilación; este espacio de memoria no cambia de tamaño durante la ejecución del programa. El **montículo** almacena datos y objetos complejos cuyo tamaño se determina durante la ejecución del programa. Este espacio de memoria puede cambiar de tamaño; es decir, puede expandir al declarar nuevos arreglos u objetos y puede disminuir cuando los recolectores de basura liberen espacio a objetos que ya no se usarán. Esto lleva a la tercera parte de memoria: la **memoria libre**. Este espacio sirve como un “amortiguador” entre el montículo y la pila. Si la memoria libre se agota, puede resultar en la caída del programa, o en la necesidad de reclutar espacio adicional. La última asignación de memoria es la **pila**, la cual sirve para recordar llamadas a funciones y los argumentos ingresados en cada llamada. Este espacio también es dinámico como el montículo. El montículo, al acumular información, crece de una forma secuencial dentro de la memoria libre, mientras que la pila crece al revés, dentro de la memoria libre (Aho et al., 2007).

Los recolectores de basura (garbage collectors) existen para detectar fugas de memoria, y desasignar los espacios de memoria que ya no se van a usar para los datos actualmente alojados allí. Existen algunos algoritmos de recolección de basura, pero se debe tomar en consideración el tiempo de ejecución, el espacio usado, el tiempo de pause y la localidad del programa al momento de desarrollar un recolector. Para la realización de programas pequeños, no es una gran necesidad la recolección de basura, ni la liberación de memoria, especialmente en sistemas modernos. Pero al momento de considerar sistemas multihilo, como servidores, que puedan correr simultáneamente múltiples programas de gran

escala, se hace muy necesario la recolección de basura para asegurar la buena eficiencia de los programas, y un manejo adecuado de memoria (Aho et al., 2007).

Al tener generado código intermedio, faltan dos pasos más de compilación, uno de ellos siendo opcional. A continuación, se verá el último paso obligatorio, la generación de código destino, lo cual recibirá de entrada el código de tres direcciones generado durante esta fase.

1.2.6 - Generación de Código Destino

Con la representación de código intermedio desarrollado, se puede generar el código destino. Esta fase tiene la obligatoriedad de llevar el mismo significado semántico del código fuente hacia el código generado; esta precisión es más importante que tener código eficiente u óptimo.

Al generar el código final, se debe tomar en cuenta cuál va a ser el nivel jerárquico del código destino. En caso de ser nivel medio o bajo, se tendrá que preocuparse por dos aspectos muy importantes: la repartición y asignación de registros. Estos pasos de bajo nivel involucran la selección de variables que residirán en los registros y en qué registros específicos estarán, respectivamente (Aho et al., 2007). En el caso de compilar a un lenguaje de alto nivel, no es necesario preocuparse por este problema.

Si la compilación se llevará a código destino de bajo nivel, entra la necesidad de ver las características de la máquina (procesador) donde el programa se ejecutará. Al saber qué formato general tiene el procesador, se debe saber cuántos registros tiene. Con este dato, se puede proseguir en la búsqueda de la manera de llenar los registros de la forma más eficiente. Este paso es esencial, al tomar en cuenta la jerarquía de memoria de una computadora: se tiene acceso inmediato a los registros, pero se requiere más tiempo, medido en ciclos de reloj del procesador, para acceder a ubicaciones de memoria más bajas en la jerarquía. Se puede demorar aproximadamente cien veces más acceder a datos de memoria RAM versus acceder a los que ya están en un registro. Cuando hay datos alojados en memoria virtual, es decir en un disco duro o disco de estado sólido, se aumenta considerablemente esta cantidad de tiempo: a 5 milisegundos o alrededor de 50 microsegundos, respectivamente. Los datos requeridos más frecuente deben estar en un registro reservado el máximo tiempo posible, y el resto de los datos se deben encontrar en el nivel jerárquico más alto posible para disminuir al máximo el tiempo de acceso a ellos, aumentando el rendimiento del programa (Aho et al., 2007).

1.2.7 - Optimización de Código

La fase de optimización de código no es imprescindible como las otras fases, y se puede desarrollar un buen compilador sin ella; pero en los casos cuando no hay muchos recursos

computacionales, conviene buscar maneras de reducir su uso. Por errores de los programadores, se puede generar código que no sea muy eficiente. (Aho et al., 2007).

Al momento de realizar una compilación, se debe construir bloques básicos y diagramas de flujo, vinculados mediante listas enlazadas. Esto permite una optimización más fácil. Técnicas de optimización local se pueden ejecutar sobre estos bloques básicos. Las técnicas buscan eliminar subexpresiones locales comunes, código “muerto” e identidades algebraicas. Al buscar subexpresiones locales comunes, se debe eliminar las instrucciones que calculan un valor que ya se haya calculado con anterioridad. Código muerto es aquello que es incondicionalmente inaccesible. Aunque este código no requiere mayor tiempo computacional, sí requiere más espacio en memoria, lo cual podría servir para otros usos. Por último, las identidades algebraicas ayudan a extirpar instrucciones inútiles. Por ejemplo, al sumar cero o multiplicar uno a un número, se obtiene el mismo número. La supresión de estas instrucciones libera memoria y reduce el tiempo de cálculos que deba ejecutar la máquina destino. Existe también técnicas basadas en el tiempo requerido de ejecución de un comando específico. Por ejemplo, elevar un número al cuadrado requiere más tiempo computacional que multiplicarlo por sí mismo, lo cual da el mismo resultado. Multiplicar un número por dos se demora más que sumar un número a sí mismo. La multiplicación por un número decimal puede resultar más fácil que su división. (ITS Education Solutions Limited 2012).

Otra especie de optimización se llama optimización de “mirilla.” Esto es el análisis sobre pequeños fragmentos de código intermedio, e implementa técnicas como eliminación de instrucciones redundantes, optimizaciones de flujo de control, simplificaciones algebraicas, reducciones de fuerza y el uso de características específicas de máquinas. La eliminación de instrucciones redundantes omite aquellas que no tienen ningún uso práctico. Un ejemplo incluye la carga de un dato a un registro y el posterior almacenamiento del registro a la ubicación de memoria de donde fue sacada. Las optimizaciones de flujo de control eliminan saltos a saltos; es decir, cuando existe un salto en el código hacia otro salto, es un flujo ineficiente. La eliminación de estos saltos redundantes mejora el rendimiento del programa. Por último, el uso de características específicas de máquina incluye el empleo de instrucciones particulares para una arquitectura específica. Un ejemplo es el incremento unitario; en vez de realizar una operación de suma, se puede ejecutar un incremento sobre el registro (Aho et al. 2007; ITS Education Solutions Limited 2012).

La optimización de código es un paso que se puede omitir en sistemas donde el uso de recursos no es crítico, y es preferible no realizarlo si hay la posibilidad de crear discrepancias entre el sentido original del código fuente y el código objeto generado. Es preferible mantener código correcto antes que código optimizado, pero erróneo.

1.2.8 - Compiladores de Lenguajes Naturales

La teoría de compiladores se puede usar en varias aplicaciones, aparte de solamente compiladores; preprocesadores, conversores fuente-fuente, rutinas de análisis de comandos y lenguaje natural (Brookshear, 1993).

Los compiladores de lenguajes naturales son un gran reto, considerado sumamente complejo y prácticamente imposible. Su ventaja es la posibilidad de programar en el idioma que uno desee, sin necesidad de usar “código”; la sintaxis tendría una estructura como una oración natural. Además, permitiría la ejecución de sentencias de muy alto nivel, que podrían traducirse a múltiples instrucciones de bajo nivel. Debido a la complejidad inherente de este problema, ha habido muy poco trabajo sobre ello, y no se ha estudiado a profundidad, como los lenguajes formales. Se realizó un estudio que extrajo del largo proceso de compilación tradicional y lo redujo a dos pasos: el *párser* y el generador de código. Para el análisis sintáctico, no se puede usar gramáticas libres de contexto, ya que “no son capaces de generar lenguaje natural completamente.” Se usó una gramática minimalista, que resultó mejor que una gramática libre de contexto, pero tampoco es suficiente para modelar completamente un lenguaje natural. Se considera la posibilidad de implementar una gramática de concatenación de rango, lo cual es más expresivo que las gramáticas minimalistas (Zúñiga, Sierra, Bel-Enguix & Galicia-Haro 2018).

1.3 - La Programación en la Educación

1.3.1 - Programación a nivel de educación básica y bachillerato

Con el paso del tiempo, se ha ido tratando de crear lenguajes de programación que acoplen a las necesidades educativas de programadores novatos. Uno de los primeros lenguajes educativos con capacidades gráficas fue Logo, lo cual fue desarrollado en el año 1967 (Kalelioğlu 2015a; Mladenović, Mladenović & Žanko 2020). Este lenguaje, con la inclusión de gráficos de tortuga, fue un gran avance para la enseñanza de la programación para estudiantes universitarios e incluso de bachillerato. En años más recientes, se ha propuesto un nuevo estilo de programación; un paradigma visual basado en bloques. Esta modalidad de programar tiene algunas ventajas, como la eliminación de errores sintácticos, y permite programar juegos, realizar animaciones, entre otras actividades multimediales (Mladenović et al., 2020). Se puede tomar en cuenta plataformas como Code (code.org), Scratch (scratch.mit.edu) y Alice (www.alice.org), como los ambientes más populares y dinámicos en el área.

Al enseñar la programación a los niños y a los preadolescentes, es muy conveniente usar, más que lenguajes de programación textuales, lenguajes de bloques. Esto ayuda a los

estudiantes a concretar los aspectos abstractos que están aprendiendo, y permite visualizar lo que están haciendo. Es menos frustrante aprender estos lenguajes, y ayuda a disfrutar mejor del aprendizaje (Ali & Smith 2014; Chen, Haduong, Brennan, Sonnert & Sadler 2019). Según Meerbaun, Armoni, Ben-Ari, Powers & Techapalokul (citado por Chen et al. 2019), hay más desventajas de aprender los lenguajes de bloques como primera modalidad de programación, como el desarrollo de malos hábitos. Por este motivo, se debe limitar la enseñanza de este paradigma, como se mencionó antes, a los niños y preadolescentes (Chen et al., 2019).

Para los estudiantes que se encuentren cursando los años de bachillerato, es más ventajoso que el estudio sea encaminado a los lenguajes textuales, permitiéndoles aprender conceptos más avanzados y poder tener mayor control sobre la computadora. (Chen et al., 2019) Además, se facilita la transición a lenguajes industriales. Según un estudio, estudiantes de bachillerato que aprendieron en ambientes basados en bloques tuvieron puntajes iniciales más altos que los estudiantes en ambiente textual, pero al realizar la transición a Java, ambas categorías tuvieron los mismos resultados (Weintrop & Wilensky 2019). Además, en el mismo estudio, los alumnos que estudiaron lenguajes textuales tuvieron más confianza en sí mismos y disfrutaron más de la clase que los estudiantes que estudiaron en base a bloques. Al final del estudio, ambos grupos de estudiantes tuvieron una cantidad parecida de errores de compilación en Java. (Weintrop & Wilensky 2019).

Los cursos universitarios de introducción a la programación tienen altas tasas de retiro, debido a la dificultad percibida (Ali & Smith 2014; Chen et al. 2019; Hawi 2010; Tuparov, Tuparova & Tsarnakova 2012; Yadin 2011). Los estudiantes que hayan estudiado la programación antes de seguir estos cursos tienen menores posibilidades de retirarse de ellos (Chen et al., 2019). Otro motivo por lo cual se debe contemplar el estudiar programación en el colegio antes de seguir una carrera universitaria enfocado a la programación, es la mejora del entendimiento de los conceptos fundamentales de ella. Según varios estudios, solo el 38% de novatos de la programación en nivel universitario pueden realizar correctamente un programa que calcule el promedio de los números, el 60% pudieron rastrear exitosamente un bucle while y el 32% pudieron predecir el valor de retorno de un procedimiento que consistía en un bucle while (Mladenović et al., 2020). Los estudiantes que ingresen a la universidad con conocimientos previos de programación tienen mejores notas y actitudes sobre el desarrollo de su carrera (Chen et al., 2019).

Desafortunadamente, debido a los prejuicios negativos hacia el tema, hay un declive de estudiantes que se matriculen en carreras afines. En general, “las mujeres en el nivel de bachillerato perciben a la informática como aburrida y expresaron una fuerte aversión a las computadoras” (Chen et al., 2019). Esto genera la posibilidad de introducir a los estudiantes

de educación general a la programación, para “combatir estereotipos de género de carreras en ciencia, tecnología, ingeniería y matemáticas (STEM)” (Chen et al., 2019).

Es necesario que los profesores que impartan temas de programación en el nivel de bachillerato lo hagan de una forma que tenga mayor enfoque en creatividad y temas multimediales, que en resolución de problemas. Al tratar de aumentar la creatividad de los estudiantes a través de temas de programación multimedial, se aumenta el interés hacia ella, y las habilidades de pensamiento crítico aumentarán como efecto. Al tratar a la programación como un tema interesante, y no como algo difícil, los estudiantes mostrarán mayor aliciente y será más probable que aprendan mejor los conceptos impartidos (Harris, 2011; Kalelioğlu, 2015).

A pesar de que ya se ha mencionado sobre las ventajas de enseñar un paradigma específico de programación a estudiantes de determinada edad, se ha mostrado que, independientemente del lenguaje aprendido, los alumnos con conocimientos en el tema tienen más posibilidades de triunfar en carreras de informática y tendrán mejores actitudes (Chen et al., 2019).

Según un estudio realizado con niños de escuela que no sabían inglés, fue difícil al inicio aprender cómo teclear palabras en inglés para programar con un ambiente textual, con el lenguaje Processing. Después de un tiempo, pudieron acostumbrarse a escribir las palabras requeridas (Tsukamoto et al., 2015). Esto muestra que no es imposible que los niños escolares aprendan cómo programar en un lenguaje textual, pero aporta a la idea que es más fácil aprender cómo escribir palabras en la lengua materna.

Muchos profesores de programación en los estudios colegiales se dedican más a la enseñanza de conceptos léxicos, sintácticos y semánticos, pero no hay tanto énfasis en la estructura y la calidad de código de los programas. Además, afirman que la estructura de código es el concepto más difícil de asimilar para los alumnos. Entre los temas de estilo de código, se encuentran documentación, presentación, algoritmos y estructura. Es necesario aprender estas temáticas para poder desarrollar código de calidad. Para que los estudiantes puedan llegar a niveles universitarios con fundamentos de programación, debe ser un requisito que sepan también cómo desarrollar código de alta calidad, con una estructura óptima y adecuada. Si se enseña estos tópicos desde el inicio, los estudiantes no tendrán problemas más adelante. Otra queja de los profesores es que no hay suficiente tiempo para dar retroalimentación a cada estudiante sobre su estilo de código; este dato indica la necesidad de más educadores de informática y clases de menor tamaño. Muchos maestros también aseveran la necesidad de tener un vínculo entre las clases de programación de bachillerato y las universidades, para que los docentes universitarios puedan ayudar en la

impartición de las temáticas más complejas (Crow, Kirk, Luxton-Reilly & Tempero 2020; Kirk, Tempero, Luxton-Reilly & Crow. 2020).

1.3.2 - Programación a nivel universitario

En el nivel universitario, muchos docentes enseñan la programación con lenguajes obsoletos, y se dedican a impartir conocimientos de la programación estructurada porque es lo que ellos aprendieron durante su formación académica. Actualmente, se considera que un lenguaje de Programación Orientada a Objetos, como Java, Python o C#, debe ser el primer lenguaje aprendido, y que no se debe demorar en presentar los conceptos POO. El usar instrumentos anticuados y obsoletos puede contribuir a una pérdida de interés de los estudiantes. Hay que recordar que, durante los años de estudio de una carrera universitaria, las tecnologías pueden cambiar, y el usar instrumentos ya arcaicos no contribuye adecuadamente al perfil profesional (Prokofyeva, Uhanova, Katalnikova, Synytsya & Jurenoks 2016).

Muchos estudiantes reprueban los cursos introductorios de programación por una deficiencia de métodos de enseñanza o por usar un lenguaje inapropiado. Según un estudio, el simple hecho de cambiar de un ambiente visual a otro redujo la cantidad de reprobación en una materia de introducción a la programación de 45,8% a 16,7%. Otra ayuda podría ser el aprendizaje en grupo, ya que los estudios sugieren que el “aprendizaje en grupo ayuda a fortalecer el entendimiento de los temas con mayor rapidez y eficiencia” (Yadin, 2011).

Otra manera de despertar el interés de los estudiantes es mediante la enseñanza de programación de juegos. En un estudio, al pedir proyectos que sean juegos, se vio que muchos alumnos comenzaron una competencia informal para ver quién podía producir el mejor juego. “Su motivación no pareció venir de su deseo de alcanzar una alta calificación, sino de un deseo de que su juego sea aceptado entre sus compañeros” (Harris, 2011). El mostrar ejemplos antes de explicar el tema también despertó el interés y aumentó la velocidad de aprendizaje en los estudiantes.

1.3.3 - Cerrando la brecha de género en la programación

Según datos en los Estados Unidos, existe una diminuta cantidad de mujeres que estudian ciencias de la computación y carreras afines. En la década de los años ochenta, 37% de las personas graduadas en la carrera eran mujeres; en el año 2013, la cantidad ha bajado a 14%. En general, las mujeres tienen menos confianza en sus habilidades de programación que los hombres (Rubio, Romero-Zaliz, Mañoso & De Madrid 2015). Pero existe la posibilidad de que esto se atribuya a la forma de enseñanza más que las habilidades de las chicas.

En un estudio realizado por Rubio et al. 2015, se usó una placa Arduino y se armó circuitos con sensores y LEDs, y la programación fue realizada en Matlab. Se usó melodías musicales para enseñar arreglos, sensores de luz y LEDs para instruir sobre estructuras condicionales, y sensores ultrasónicos y un servomotor para reforzar el tema de bucles. Con el grupo de estudiantes que estudiaron mediante este método, no hubo diferencias significativas entre los hombres y las mujeres sobre su percepción de la facilidad y utilidad de la programación. En el grupo de control, que utilizó métodos tradicionales de enseñanza de programación, sí hubo una gran diferencia entre las percepciones de hombres y mujeres. El grupo de control tuvo 35% de reprobación entre las mujeres y 17% en los hombres. Pero en el grupo experimental, era 19% para mujeres, y 18% para los hombres. Otros estudios han mostrado que la implementación de música y multimedia en la programación aumenta el interés y el aprendizaje de las mujeres en la programación (Rubio et al., 2015). Estos datos muestran que las mujeres son igualmente capaces en la programación que los hombres, pero se requiere de una nueva forma de enseñanza que sea más atractiva, mediante la implementación de diferentes medios.

CAPÍTULO 2

Desarrollo

2.1 - Introducción

El desarrollo del proyecto se lo realizó en base a lo estudiado durante la realización del marco teórico. Se ha decidido que el compilador a desarrollar debe tener un equilibrio entre sencillez de uso, funcionalidades de interés para los estudiantes y potencia para el aprendizaje de diferentes fundamentos de la programación. Además, se tomó en cuenta la necesidad de versatilidad y el poder hacer cambios al lenguaje sin necesidad de cambiar drásticamente al código.

Los datos requeridos para el autómata finito determinista y el autómata de pila, para el explorador y el reconocedor respectivamente, no están fijados en el código, sino que se cargan desde archivos de texto, permitiendo modificaciones rápidas.

Se incluyó dentro del compilador un actualizador automático que permite buscar actualizaciones en línea, y descargárselos automáticamente sin intervención del usuario. También se implementó un cliente de correo y posteriormente una base de datos en Azure para el envío de código y posterior análisis.

2.2 – Recursos y Materiales

2.2.1 - Materiales

- Una computadora de escritorio con procesador i5 10ma generación, 16gb de memoria RAM DDR4, disco de estado sólido M.2 480gb y tarjeta gráfica GeForce GTX 1650 con Windows 10 instalado.
- Licencia de Windows 10.
- Dos monitores; uno para desarrollo, otro para pruebas e investigación.
- Teclado y mouse USB.
- Parlantes para probar funcionalidades auditivas.
- Otra computadora de escritorio con procesador Intel Pentium, 16gb de memoria RAM DDR3, disco duro de 500gb y tarjeta gráfica GeForce GTX 1650 con Windows 10 instalado.

2.2.2 - Software Utilizado

- Para el diseño del programa, incluyendo la interfaz y el código que realiza todas las funcionalidades del compilador, se decidió utilizar Visual Studio 2019 y el lenguaje C#, por su facilidad y amplitud de características.
- Para generar el autómata inicial de la fase léxica, se utilizó Sefalas. A pesar de los “bugs” que tiene, es un programa fácil de usar y permite una generación rápida del autómata finito determinista requerido.
- Se intentó usar Sefalas también para generar el autómata de pila del analizador sintáctico, pero debido a la complejidad del lenguaje, la herramienta no tuvo capacidad suficiente. Se decidió recurrir a la herramienta en línea de JSMachines.
- En la compilación de programas gráficos, se usó XNA⁴⁰ 4.0.
- Para la creación de diagramas especificando la gramática, se usó la herramienta en línea llamada draw.io.
- Se decidió incorporar actualizaciones automáticas al programa. Para alojar el archivo .xml de la versión más actual, se usó la plataforma gratuita 000.webhost.com.
- La plataforma usada para almacenar las últimas actualizaciones es Dropbox.
- Al inicio de la evaluación, se usó una cuenta de correo no institucional, en Gmail, para recibir y evaluar código escrito por usuarios.
- Posteriormente se recurrió al uso de una base de datos alojada en Azure para recibir los resultados de código compilado por usuarios.

2.3 – Metodología

Para el desarrollo del proyecto, se utilizó la metodología de Cajas de Christopher Jones para tener una transparencia en los objetivos, y a la vez se oculta al usuario los procedimientos seguidos para no complicar su uso del programa. Se adhirió también a las normas aplicables de MISRA-C para tener un código más legible y seguro.

El compilador desarrollado es de una pasada, que permite detección de errores rápidamente y un rendimiento mayor. Se inicia por el reconocedor, que solicita al explorador los tokens. Cada vez que se reconoce una regla, se invoca al analizador semántico, que corre rutinas semánticas sobre el código ingresado, produciendo el producto final, que es el

⁴⁰ **XNA** – Un framework de Microsoft que permite el desarrollo de juegos 2D y 3D para computadoras Windows y la consola de videojuegos Xbox.

código en C#. Tras una compilación inicial exitosa, se procede a ejecutar el compilador C# de Microsoft. Este convierte el código compilado hasta el momento en un archivo ejecutable.

Cada fase de análisis fue separada como su propia clase, con los métodos y parámetros requeridos. En lo posible, se ha limitado la interacción entre clases, para que el programa sea más modular.

A continuación en la figura 5, se detalla en forma gráfica la arquitectura general seguida para el desarrollo del programa, con sus respectivas entradas, salidas, controles y mecanismos:

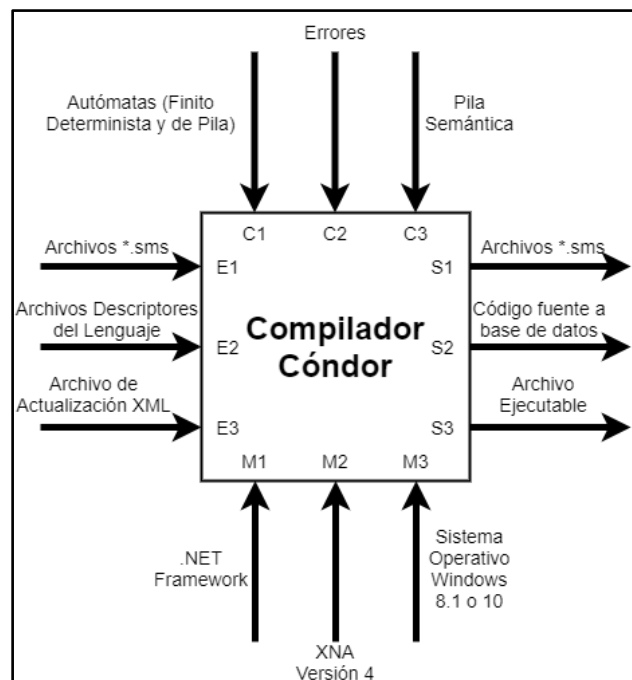


Fig. 5. Arquitectura del programa con entradas, salidas, controles y mecanismos.

Desarrollando esta caja general y global del programa, se puede analizar cada componente o módulo principal del proyecto, indicando las partes que se interconectan y el flujo de información, mostrado en la figura 6.

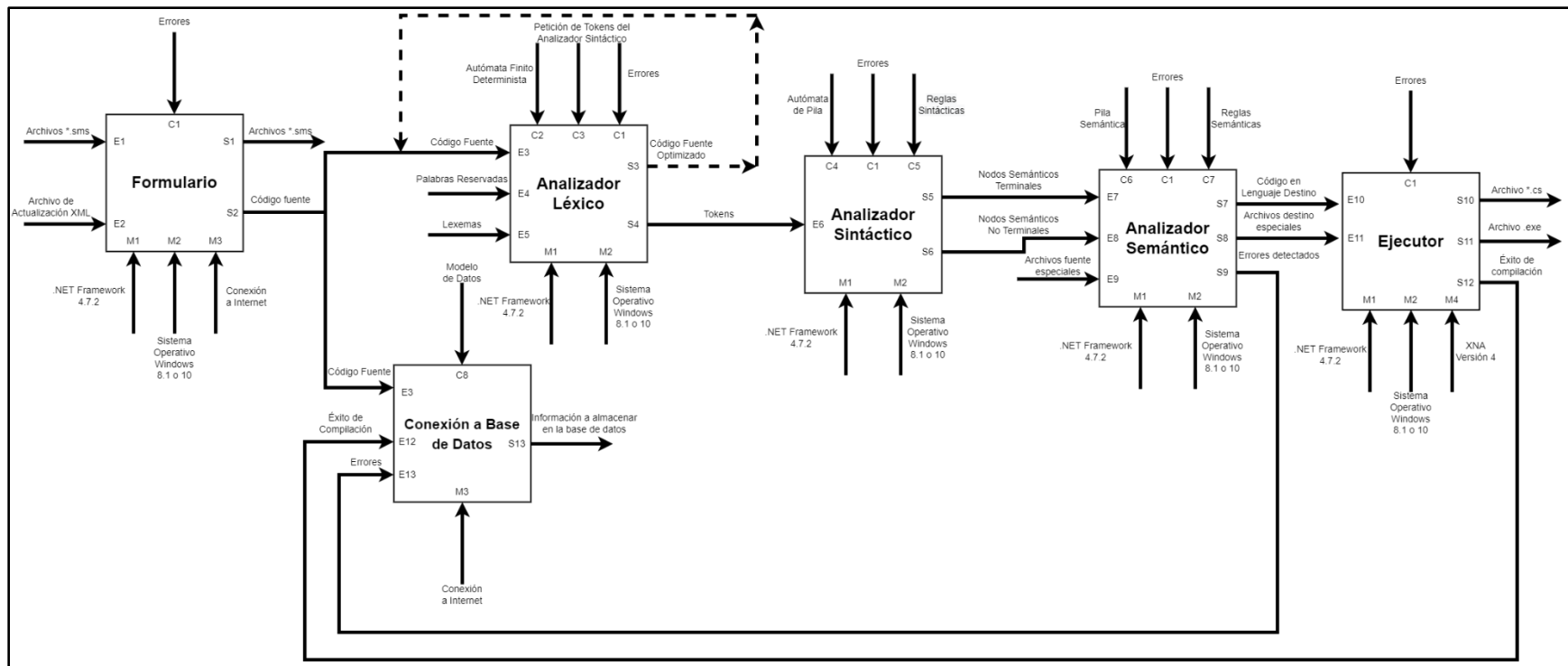


Fig. 6. Interconexión de los diferentes componentes del programa y sus entradas, salidas, controles y mecanismos.

Además, es necesario especificar el flujo de información en cada proceso principal del programa. La figura 7 muestra los símbolos usados en los siguientes diagramas.

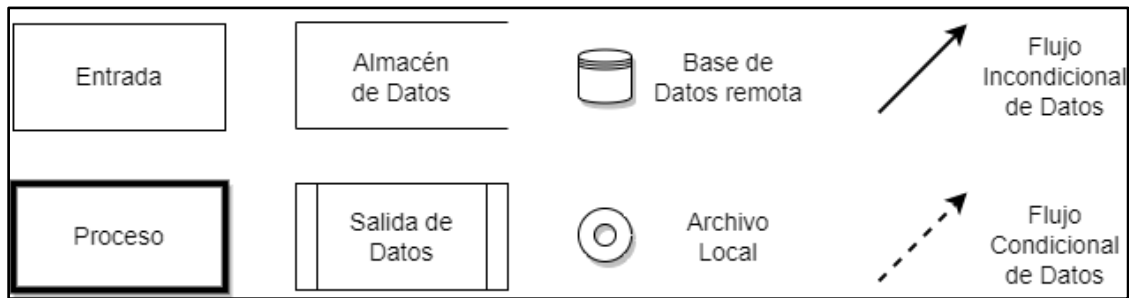


Fig. 7. Leyenda de símbolos usados en los diagramas de flujo de información.

El primer proceso que se ejecuta al mostrar la interfaz del compilador es la actualización del programa, como se puede apreciar en la figura 8.

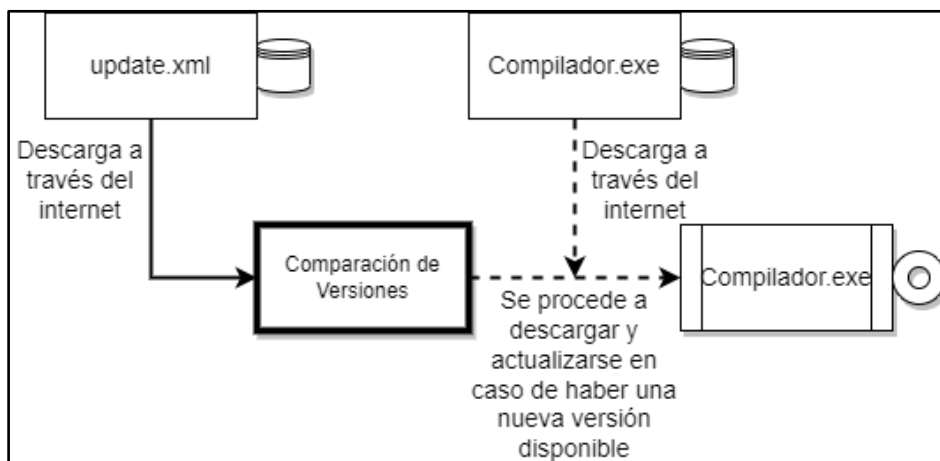


Fig. 8. Diagrama de flujo de información durante el proceso de actualización.

Al arrancar el programa, el compilador recibe el identificador del usuario desde la base de datos en Azure. En cada compilación, se escribe el código fuente en la base de datos. También se escriben los errores encontrados. Se indica este proceso en la figura 9.

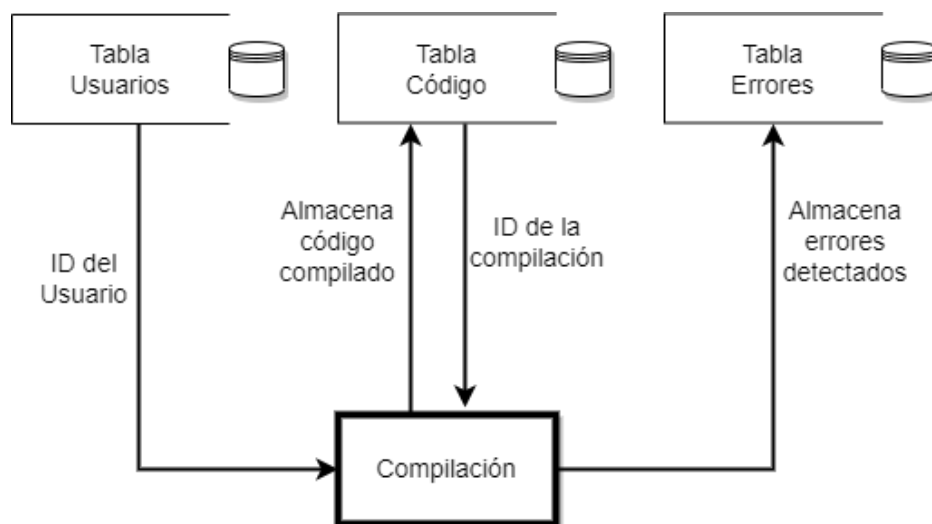


Fig. 9. Diagrama de flujo de información durante el proceso de envío de datos a la base de datos.

El proceso de abrir y guardar archivos es muy sencillo; se pueden abrir y guardar archivos con la extensión .sms (Spanish Multimedia Source), mostrado en la figura 10. Los archivos se guardan de forma textual y se pueden abrir con cualquier editor de texto.

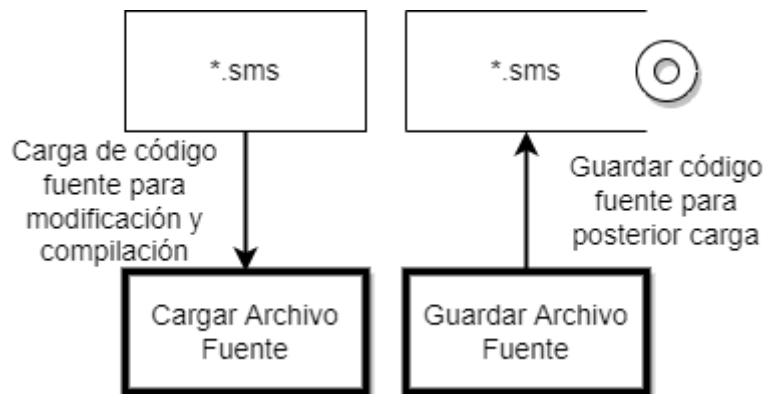


Fig. 10. Diagrama de flujo de información durante el proceso de lectura y escritura de archivos de código fuente.

Cuando se hace un análisis o escaneo del código fuente escrito en la pantalla, pasa por los analizadores léxico, sintáctico y semántico, mostrando los errores hallados. La figura 11 muestra estos pasos.

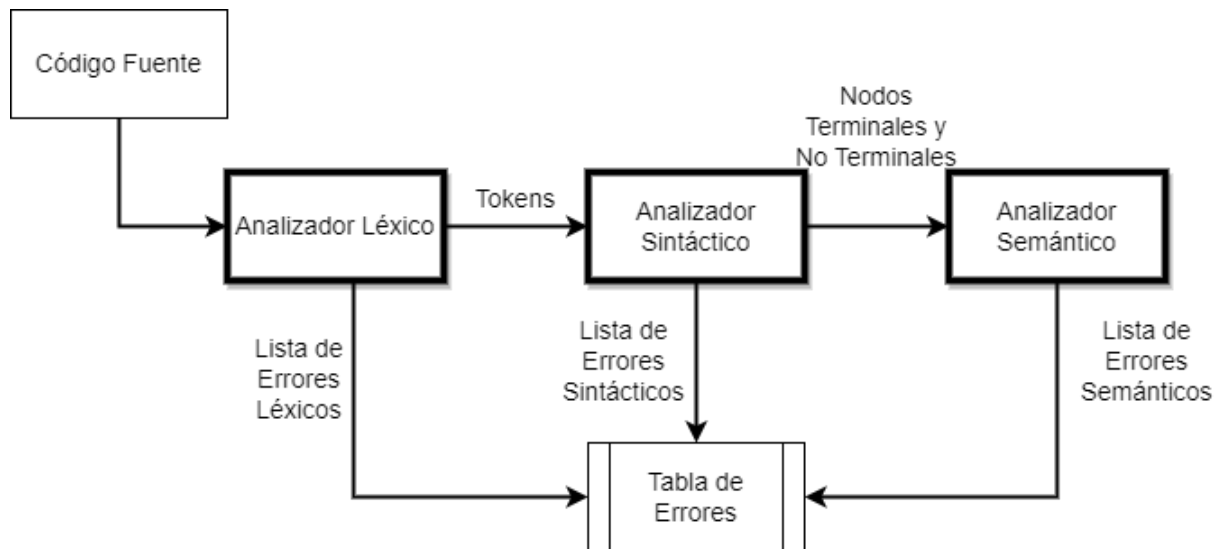


Fig. 11. Diagrama de flujo de información durante el proceso de análisis de código.

Una compilación es más compleja. El código es enviado a la base de datos, los errores generados son exportados también a ella y el código generado durante el análisis semántico es escrito a un archivo .cs y compilado a un archivo ejecutable. La figura 12 indica este proceso.

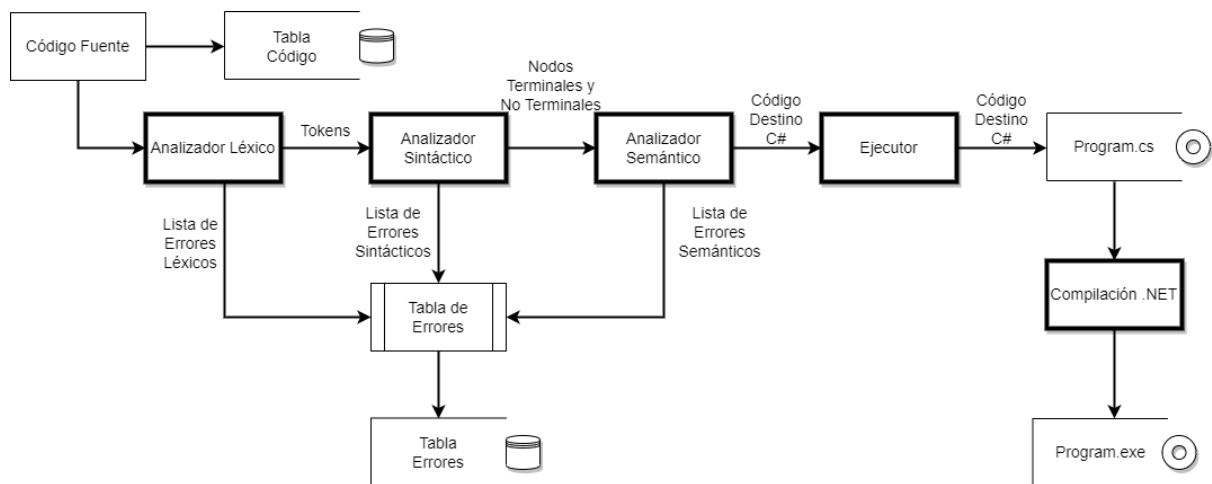


Fig. 12. Diagrama de flujo de información durante el proceso de compilación.

El compilador fue desarrollado siguiendo las especificaciones ya mencionadas, desarrollando cada módulo separado de los demás, con un avance secuencial desde el formulario hasta la conexión a la base de datos.

Tras haber completado un nivel adecuado del programa, se procedió a distribuirlo. Durante las primeras semanas de uso, se hizo mejoras iterativas a cada módulo, con la intención de aumentar el rendimiento del código, tener mejor documentación, reducir la cantidad de errores, etcétera.

Hay documentación extensiva en el código fuente del compilador; cada clase, variable, constructor y método tiene comentarios que explican de forma detallada su uso y funcionamiento.

Los siguientes estándares fueron seguidos durante la creación del código:

- Ningún comentario contiene código – solamente descripciones y explicaciones.
- Ningún identificador excede 31 caracteres.
- El operando de lado derecho de operadores lógicos no ejecuta sentencias no condicionales.
- Se ha colocado paréntesis en condiciones compuestas y alrededor de los operandos de cada operador relacional.
- No hay código inalcanzable.
- No se ha usado la sentencia *goto*.
- Todas las instrucciones *switch*, *while*, *do-while*, *for* e *if* tienen llaves que delimitan su inicio y final, aunque haya solo una instrucción.
- La sentencia *break* termina cada cláusula de cada *switch*.
- Ninguna función se llama a sí misma – no hay recursividad.

- Los nombres de las variables empiezan con una letra minúscula y tienen letras mayúsculas al inicio de cada palabra a partir de la segunda.
- Los nombres de las constantes se encuentran completamente en mayúsculas.
- Los nombres de las clases y los métodos tienen una letra mayúscula al inicio de cada palabra.
- Se emplea *const* para constantes simples cuyo valor se conoce al momento de compilación.
- Se emplea *readonly* para constantes complejos cuyo valor se conoce al momento de compilación.
- Variables no usadas para lectura fuera de la clase son de tipo *private*, con un método de escritura de tipo *public*.
- La realización de tareas adicionales fuera de la compilación normal se realizará en otros hilos para no detener la ejecución del programa. Específicamente, esto aplica para la actualización y el envío de información a la base de datos. (MIRA Limited, 2004; Motor Industry Software Reliability Association, 2016)

Con los materiales y software requeridos, metodología, arquitectura y estándares previamente descritos definidos desde el inicio, se pudo continuar con el desarrollo del proyecto.

2.4 – Especificación del lenguaje

Para el desarrollo del lenguaje se decidió que, en lo posible, debe ser sencillo para que los estudiantes de bachillerato puedan aprender con mayor facilidad. Se usan los operadores aritméticos más básicos, los identificadores (nombres de variables) comienzan con una letra y son seguidos por otras letras y números. Se permite la escritura de tildes y eñes en el código, posibilitando una ortografía correcta, que no se puede tener en lenguajes modernos.

Se incluye el reconocimiento de cadenas de texto y literales de tipo numérico (entero y decimal), carácter y lógico. Además, se permite la identificación de literales de tipo color. Se ha excluido los operadores lambda y de bits ya que es un lenguaje introductorio.

Las palabras reservadas se hallan dentro de la categoría de “identificadores”, reduciendo considerablemente el tamaño del autómata (se explicará este concepto en mayor detalle posteriormente). También se ha programado algunas funciones que trabajan con entradas y salidas.

Los diferentes aspectos del lenguaje, incluyendo el léxico, las palabras reservadas, las funciones preprogramadas, las reglas gramáticas y las reglas de operadores se adjuntan en la sección de anexos.

2.5. – Desarrollo de la Interfaz Gráfica

2.5.1 – Arquitectura de la Interfaz Gráfica

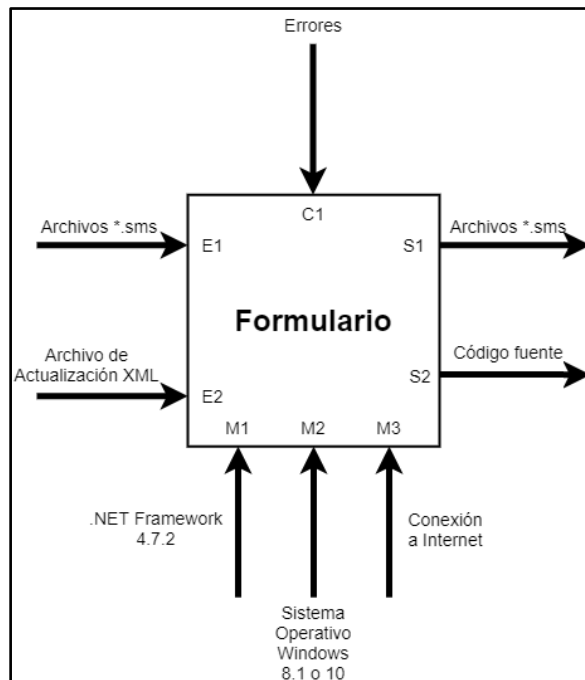


Fig. 13. Arquitectura de la interfaz gráfica.

La ventana de la interfaz gráfica permite cargar y guardar archivos *.sms, que contienen código fuente. El archivo de actualización .XML se obtiene mediante una conexión a Internet, y sirve para buscar nuevas actualizaciones del programa. Al momento de realizar una compilación, el código fuente es pasado a la fase siguiente del programa. La figura 13 muestra las entradas, salidas, controles y mecanismos usados por el formulario.

2.5.2 – Apariencia y Funcionalidades Básicas

La interfaz del programa fue realizada de una manera sencilla para que el usuario no tenga que aprender muchas funcionalidades y pueda comenzar a escribir código rápidamente. Existe la implementación de funciones básicas de manejo de archivos, opciones básicas de compilación y una sección de ayuda que incluye manuales de usuario, ejercicios resueltos y propuestos, y ejemplos de código.

Antes de ver la interfaz principal, se muestra una pantalla de bienvenida, mostrada en la figura 14, que indica el logotipo del compilador y algunos datos adicionales, como los autores y la versión actual.



Fig. 14. Pantalla de bienvenida del compilador donde se indica su versión, creador, asesor y la versión.

Después de cuatro segundos se puede presenciar el formulario principal del programa, como se muestra en la figura 15.

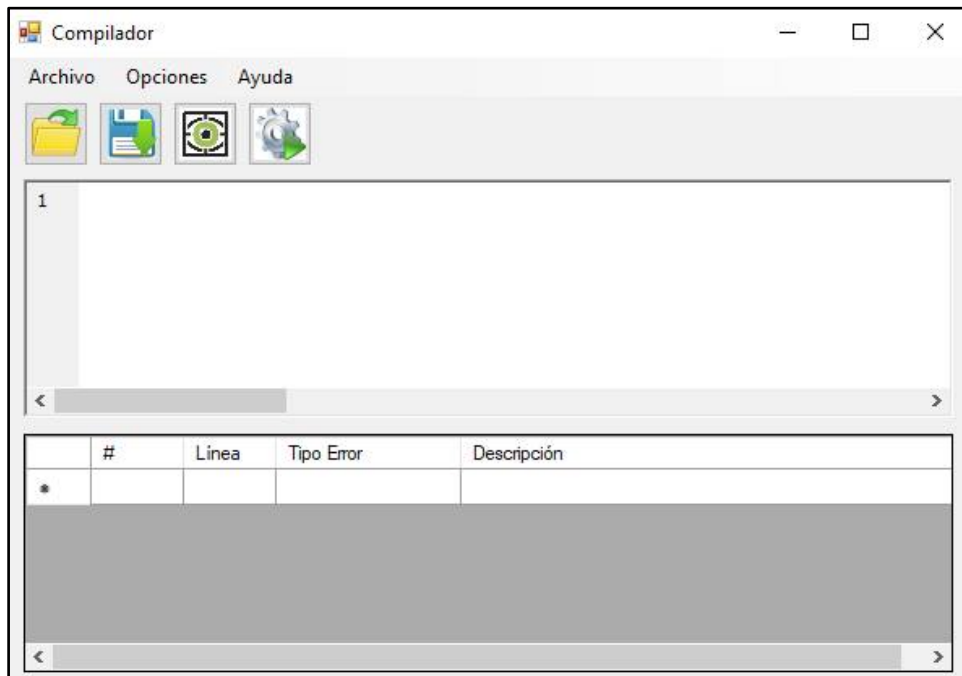


Fig. 15. Interfaz del compilador.

Al teclear código, cada lexema se va tornando de un color diferente dependiendo de su categoría, como se detalla a continuación:

- Comentarios – Verde
- Cadenas de caracteres – Rojo

- Caracteres – **Morado**
- Identificadores (nombres de variable) – **Celeste**
- Operadores – Negro
- Números – **Amarillo**
- Palabras reservadas (tipos de datos, palabras de estructura de control y bucles, etc.) – **Azul**
- Palabras reservadas (literales y funciones preprogramadas) – **Tomate**
- Tiras no reconocidas – **Negro** (de tamaño inferior y subrayadas de rojo)

Los botones y opciones de menú tienen las siguientes funcionalidades de la figura 16:

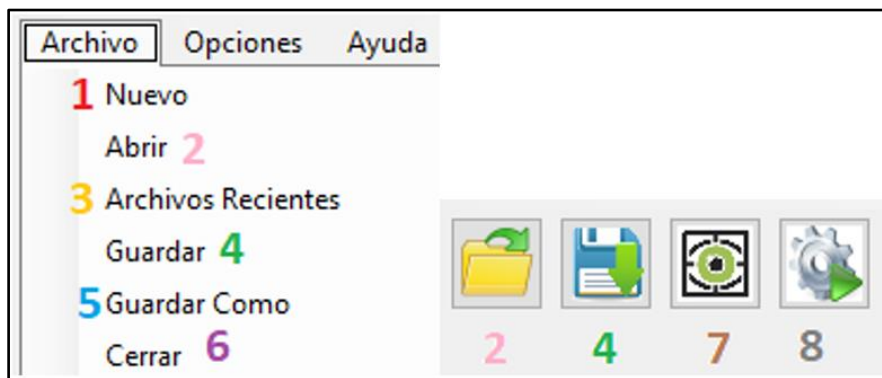


Fig. 16. Menú de funciones desplegadas tras seleccionar "Archivo" y funciones principales.

1. **Nuevo** - Limpia todo el código ingresado en el cuadro de texto, y reinicia la ruta del archivo actual (es decir, al dar guardar, pedirá la ubicación y nombre deseados).
2. **Abrir** - Abre un archivo de código fuente (.sms).
3. **Archivos Recientes** - Muestra una lista de los archivos recientemente abiertos (hasta diez). Al hacer clic en uno de ellos, se abrirá inmediatamente.
4. **Guardar** - Guarda el código fuente en un archivo .sms. Si se ha abierto o guardado previamente un archivo desde la ejecución del programa, guardará automáticamente en la ubicación de ese archivo.
5. **Guardar Como** - Guarda el código fuente en un archivo .sms. Independientemente de la ruta del archivo actual, pedirá la ubicación y nombre deseados para guardar el código actual.
6. **Cerrar** – Cierra el programa.
7. **Escanear** – Escanea el código para ver si hay errores, que se mostrarán en la tabla de errores.
8. **Compilar** – Escanea el código, y si no contiene errores, crea un programa ejecutable acorde al código del cuadro de texto, y lo ejecuta.

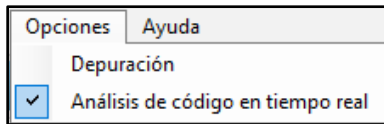


Fig. 17. Menú de funciones desplegadas tras seleccionar "Opciones".

- **Depuración** – Cuando esta opción está seleccionada, permite ver en la estructura de árbol los diferentes valores que pueden tener las variables durante la ejecución de un programa. Esta funcionalidad solo se puede ejecutar en modo consola, es decir, que no funciona en modo gráfico. (Opción desactivada por defecto)
- **Análisis de código en tiempo real** – Al tener activada esta opción, cada teclada inicia (o reinicia) un temporizador de tres segundos. Al culminar este lapso de tiempo, se hace un análisis del código para ver si existen errores, que se mostrarán en la tabla de errores. (Opción activada por defecto)

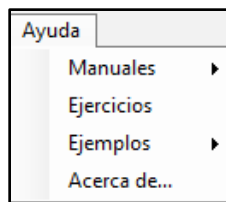


Fig. 18. Menú de funciones desplegadas tras seleccionar "Ayuda".

- **Manuales** – Aquí hay dos manuales que explican el funcionamiento del compilador y del lenguaje.
- **Ejercicios** – Este botón abrirá un documento que muestra cómo desarrollar tres programas sencillos y retará al usuario a hacer tres ejercicios más de mayor complejidad.
- **Ejemplos** – Bajo este menú, podrá ver algunos ejemplos de programas que puede hacer. Usted puede modificar el código y al guardar, se guarda aparte, permitiéndole agregar giros creativos a los ejemplos ya existentes, sin preocuparse de que estos se pierdan.
- **Acerca de...** - Esta opción muestra unos datos del programa, como el creador y la versión.

 A screenshot of a code editor window. The top part shows code with error markers: a green square on line 1, a red 'E' on line 2, and a red '3' on line 5. Below the code is a table of compiler errors.

#	Línea	Tipo Error	Descripción
▶ 1	1	Sintáctico	310 Se espera :: Otro token hallado.:
2	2	Semántico	403 El token y operador usados no son compatibles con el tipo texto.: "Hola Mundo!"-"Mundo"
3	5	Semántico	414 Los argumentos ingresados no cuadran con los parámetros permitidos de la función.: HolaMundo(5)
*			

Fig. 19. Tabla de errores del compilador.

- **#** - Esta columna muestra números secuenciales, dando un número distinto a cada error hallado en el código.
- **Línea** – Esta columna muestra la línea donde el error fue hallado.
- **Tipo Error** – Esta columna indica si el error es de tipo léxico, sintáctico o semántico.
- **Descripción** – El motivo de esta columna es indicar por qué se halló un error, y en lo posible muestra el código “ofensivo”.

Mientras se vaya tecleando, aparece la ventana de autocompletado. Para agilizar el proceso de ingreso de código, se puede presionar la tecla *Intro* para completar la palabra seleccionada. Existe la posibilidad de usar las flechas arriba y abajo para seleccionar la palabra reservada deseada.

2.5.3 – Funcionalidades Adicionales

Otra funcionalidad que permite un desarrollo más ágil de código es la inserción rápida de código, como estructuras de control y bucles. Para usar esta característica, se debe poner el cursor de texto donde necesita insertar el código. Al hacer clic derecho dentro del cuadro de texto, se presenta una lista de opciones entre las estructuras de control y bucles. Se insertará un ejemplo del código deseado tras seleccionar la opción deseada y hacer clic en “Seleccionar”. Esto permite tener menos errores sintácticos y generar código con mayor rapidez.

El compilador no permite una depuración como tal, pero sí permite al final de la ejecución visualizar todos los valores que hayan tenido las variables. Por el momento, esta opción solo funciona para programas de consola, y no para los programas gráficos. Al tratar de compilar un programa gráfico con la opción “depuración” seleccionada, generará errores, debido a que un programa XNA no puede también correr un formulario.

Cada vez que se ejecuta el compilador, este buscará automáticamente (y sin advertir al usuario) actualizaciones en línea. Si hay una versión nueva del programa, se la descargará y el programa será actualizado. Esto suele suceder dentro del primer minuto de usar el programa. El programa no se detiene durante la búsqueda de una nueva versión ni durante su descarga, pero si hay una actualización, tras su respectiva descarga, el programa desactualizado se cierra y se ejecuta la nueva versión. De esta manera, el usuario no necesita preocuparse por las actualizaciones, y el desarrollador puede enviar cambios fácilmente.

2.6 – Desarrollo del Analizador Léxico (Explorador)

2.6.1 – Arquitectura del Analizador Léxico

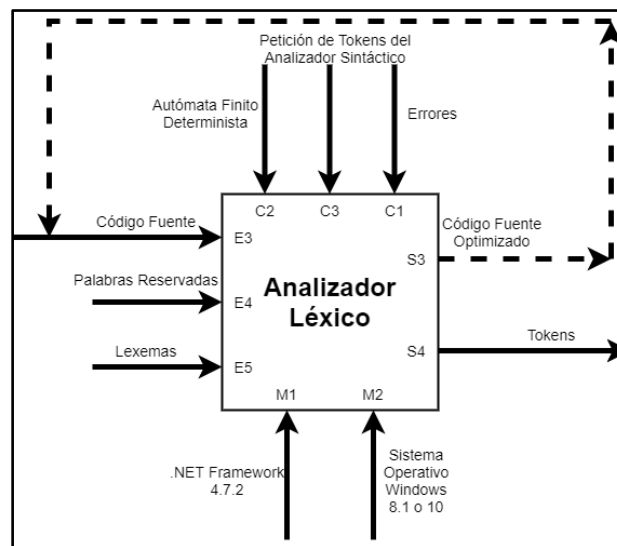


Fig. 20. Arquitectura del Analizador Léxico del compilador.

El Analizador Léxico requiere de una serie de archivos para su funcionamiento, como el autómata finito determinista, las palabras reservadas y los lexemas. Cuando llega el código fuente al explorador, este realiza una optimización previa eliminando varios caracteres para tener una compilación más rápida. Luego, el analizador sintáctico empieza a pedir tokens, que el explorador devuelve, uno a la vez. Se puede observar en la figura 20 las partes del analizador léxico.

2.6.2 – Creación y Optimización del Autómata Finito Determinista

Para desarrollar el analizador léxico, primero fue necesario crear el autómata finito determinista que ayudaría a determinar la validez de las palabras ingresadas en el código fuente. Se utilizó la herramienta gratuita Sefalas para crear el AFD. Tras especificar los caracteres permitidos en el alfabeto y las palabras aceptadas en el léxico del lenguaje, Sefalas generó la tabla de transiciones del AFD, mostrada en la figura 21:

Automata generado											
	a	á	b	c	d	e	é	f	g	h	i
0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1
2											
3											
4											
5	5	5	5	5	5	5	5	5	5	5	5
6	25	26	25	25	25	25	25	25	25	25	25
7											
8											
9											

Fig. 21. Tabla de transiciones del AFD generada por Sefalas.

Se pasó cada celda a una hoja de cálculo. Hubo un total de 36 estados, con 104 entradas. Usar el AFD de esta forma hubiese requerido 3,744 unidades de memoria. Se optó por reducir las entradas a 33; esto se realizó combinando todas las letras en una categoría, y los números uno a nueve en otra (cero se mantiene aparte). Esta reducción deja un total de 1,188 unidades de memoria requerida, al usar el AFD de esta forma. La figura 22 muestra esta simplificación.

Estado	letra	0	1	2	3	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
0	1	2	3	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38		
1	1	1	1																																				
2																																							
3			3	3																																			
5	5	5	5	24	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	
6	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	
7			27	3				28																															
8			27	3					29																														
9																																							
10																																							
11																																							
12																																							
13																																							
14																																							
15																																							
16																																							
17																																							
18																																							
19																																							
20																																							
21																																							
22																																							
23			34	34																																			
24																																							
25								36																															
27																																							
28																																							
29																																							
30																																							
31																																							
32																																							
33																																							
34			34	34																																			
36																																							
37																																							
38																																							

Fig. 22. Tabla de transiciones pasado a Excel tras reducir las entradas de 104 a 33.

Este autómata contiene muchos estados terminales sin ninguna salida. Los estados terminales sin salida son aquellos que no tienen ninguna transición para salir de ellos. El tratar de cargar esto a memoria ocuparía mucho más espacio de lo requerido, y podría generar algunas confusiones o requerir cálculos adicionales para la determinación de qué lexema fue reconocido. Todos los estados terminales sin ninguna lectura permitida fueron eliminados, dejando un total de 16 estados. Con esta nueva configuración, se requiere solamente 528 unidades de memoria para almacenar la tabla entera. Todas las transiciones que ahora llevan a estados inexistentes fueron reemplazadas por números negativos que corresponden al código del lexema detectado. La figura 23 muestra esta simplificación.

Estado	letra	0	1	2	3	5	6	7	8	9	-4	-5	-6	-7	-8	-24	-25	-22	-21	18	19	-23	21	22	-19	-20
0	1	2	3	5	6	7	8	9	-4	-5	-6	-7	-8	-24	-25	-22	-21	18	19	-23	21	22	-19	-20		
1	1	1	1																							
2																										
3			3	3																						
5	5	5	5	-28	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	
6	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	
7			27	3					-17																	
8			27	3						-18																
18																										
19																										
21																										
22																										
23			34	34																						
25																										
27																										
34			34	34																						

Fig. 23. Tabla de transiciones tras reducir los estados de 36 a 16.

conveniente para las cadenas de caracteres que pueden tener cualquier cantidad de símbolos de entrada. Cuando el autómata llegue a un estado negativo, significa que se ha reconocido un lexema. Al reconocer un lexema, en caso de requerir el atributo, se recopila todos los caracteres existentes entre “lexemeBegin” y “forward” y los guarda en el atributo del token.

Si un identificador fue reconocido, requiere un par de procesos adicionales: se debe cambiar todos los caracteres especiales del alfabeto español a caracteres similares en el alfabeto inglés. Además, si se reconoce que el identificador es una palabra reservada, se lo asigna el símbolo correcto y ya no se lo trata como un identificador.

En el caso de detectar un lexema irreconocible, el explorador simplemente lo omite y sigue con el análisis del código, previniendo la detención del analizador por una simple falla. Sin embargo, sí se crea un error y se muestra al usuario, para que el usuario sepa en qué ha fallado y por qué el token no fue reconocido.

Al culminar de analizar todos los lexemas del texto fuente, se entrega un token final, de símbolo ‘\$', delimitando el final del flujo de código. Este símbolo es reconocido como el carácter delimitador del proceso léxico.

2.7 – Desarrollo del Analizador Sintáctico (Reconocedor)

2.7.1 – Arquitectura del Analizador Sintáctico

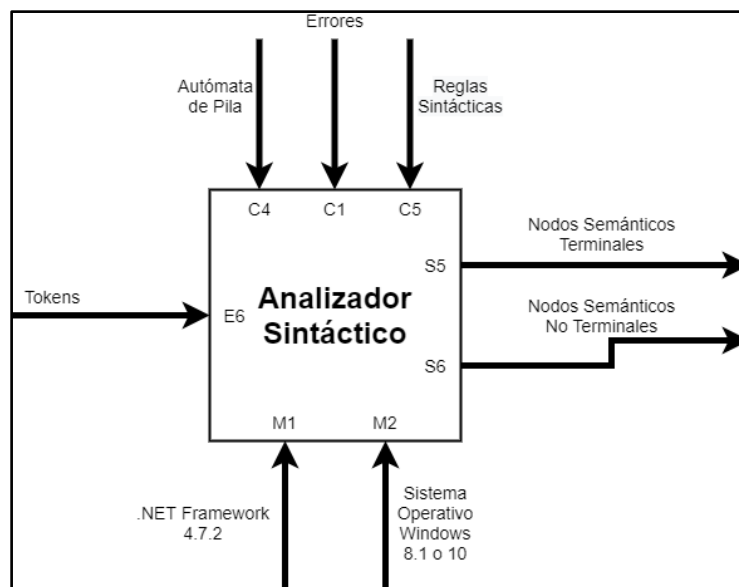


Fig. 25. Arquitectura del analizador sintáctico del compilador.

El reconocedor requiere de diferentes archivos para cargar el autómata de pila y las reglas sintácticas requeridas para validar la entrada de tokens devueltos por el explorador. Los tokens son convertidos a nodos semánticos terminales, y las reglas reconocidas se convierten en nodos semánticos no terminales. La figura 25 muestra el analizador sintáctico.

2.7.2 – Creación del Autómata de Pila

Durante la creación de la gramática del lenguaje, hubo varios inconvenientes, entre ellos el uso de una gramática ambigua y la generación de conflictos al momento de crear el autómata de pila. Al inicio se usó la herramienta Sefalas, pero se llegó a la conclusión que la herramienta no tenía la capacidad suficiente para el lenguaje. En su lugar, una herramienta en línea de JMachines fue utilizada debido a su rendimiento y facilidad de uso.

Un analizador ascendente de tipo LALR(1) fue usado para construir el AP. Existe un total de 234 estados de transición, con 55 entradas terminales en la tabla “Acción” y 40 entradas no terminales en la tabla “Ir-a”.

Los valores de las tablas “Acción” e “Ir-a” se almacenan en dos archivos separados por coma. Además, hay otro archivo que contiene todas las reglas almacenadas, también separado por coma.

2.7.3 – Funcionamiento del Analizador Sintáctico

El analizador sintáctico es el núcleo del compilador, debido a que es la primera fase e invoca a los otros analizadores en tiempo real cuando se los necesite.

Tras la limpieza de código, se ejecuta el analizador sintáctico. Cada vez que requiera un token, pide al analizador léxico. Al leer un token, existen cuatro posibilidades:

- 1) **Desplazamiento** – Almacena el token leído en la pila y pide el siguiente token.
- 2) **Reducción** – Saca elementos de la pila de acuerdo con la regla reconocida.
- 3) **Error**
- 4) **Aceptación**

En caso de error, si existe un caso de fallo fácil de recuperar, se procede a corregir el error automáticamente. Por ejemplo, si se espera el símbolo de dos puntos, la rutina de manejo de errores “ErrorRoutine” puede insertar este token requerido. A pesar de que sí muestra el error en la tabla de errores, tras su corrección automática, se puede seguir analizando el código. En un caso de recuperación de mayor complejidad, se termina el análisis en ese punto, y se muestra el error irrecuperable en la tabla. Este procedimiento permite que haya errores leves en el programa y que el usuario no siempre tenga que arreglar pequeñas inconsistencias en el código. Recuperación a nivel de frase es el método de manejo de errores empleado para algunos errores fáciles de corregir. Para los otros errores, no se usa ninguna corrección; se detiene el análisis, entregando un mensaje de error.

2.8 – Desarrollo del Analizador Semántico

2.8.1 – Arquitectura del Analizador Semántico

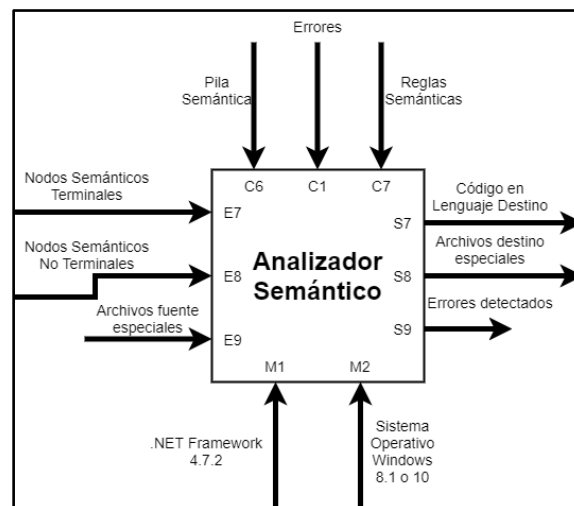


Fig. 26. Arquitectura del analizador semántico del compilador.

El analizador semántico, mostrado en la figura 26, usa una pila para almacenar los nodos terminales y no terminales entregados por el reconocedor. Con los nodos, la pila y rutinas semánticas, se va creando el código en el lenguaje destino (C#) mediante un análisis ascendente del código. Para depuración y programas gráficos, se requiere de código fuente especial cargado al momento de hacer la compilación. Después de completar esta fase, se puede devolver todos los errores detectados hasta el momento.

2.8.2 – Funcionamiento del Analizador Semántico

El analizador semántico tiene varias partes y tiene mayor complejidad que el resto del programa. Cada vez que se ejecuta el proceso de compilación (o escaneo), se reinician las variables de la clase semántica a sus valores iniciales, y las listas se vacían.

Cuando el analizador sintáctico recibe un token del analizador léxico, lo pasa al analizador semántico. Este hace las transformaciones necesarias (por ejemplo, aplica un tipo de dato si es necesario o cambia el código del token al correcto) y coloca los tokens sobre la pila semántica.

En el momento que el reconocedor detecta reglas, el analizador semántico realiza rutinas semánticas dependiendo de la regla gramatical reconocida. Cada regla tiene una definición dirigida por sintaxis, especificando cómo pasar los atributos de nodos inferiores del "árbol" (nunca se crea un árbol explícito, pero el manejo de la pila lo simula) hacia nodos superiores. Esto permite tener una traducción por atributos sintetizados. La mayoría de las rutinas suben el código de nodos inferiores hacia superiores, haciendo también verificaciones de tipos de datos. Para determinar la compatibilidad de tipos de datos, se usa el método

“CheckType”, que tiene los códigos de los tipos de datos usados y el operador como entradas. Si existe compatibilidad entre los tipos de datos y el operador, se devuelve un código que especifica el tipo de dato resultante. Si no hay compatibilidad, entonces se devuelve un número negativo que representa un error.

El analizador semántico es la única fase que tiene comunicación con la tabla de símbolos. Este agrega variables, arreglos y métodos a la tabla de símbolos para su posterior validación. Existe también un verificador rudimentario que comprueba que las variables hayan sido declaradas una vez dentro de su alcance, antes de ser usadas. También hay una funcionalidad que permite envolver código potencialmente “peligroso” con bloques *try-catch* indicando qué instrucciones dan errores durante la ejecución de un programa, dando la posibilidad de seguir con la ejecución en vez de detener por completo el programa. En el caso de generar errores durante la segunda compilación debido a este intento de prevenir errores, se recompila automáticamente sin esta funcionalidad. Cuando se hace una compilación con modo de pruebas de escritorio activado, se genera código adicional para almacenar en una estructura de árbol los valores de las variables durante la ejecución del programa.

Cuando se haya leído todo el código, y se esté llegando a los nodos superiores del árbol sintáctico (implícito), se agrega las instrucciones requeridas para completar el programa. En el caso de ser un programa de consola, simplemente se une las referencias necesarias al inicio, se crea un método *main*, y se envuelve el código generado dentro de llaves. Cuando el programa requiere una interfaz (para pruebas de escritorio) o ventana gráfica (para aplicaciones multimediales XNA), se debe leer una plantilla vacía del código fuente e insertar el código dentro de la ubicación correcta.

2.9 – Ejecutor

Cuando se termina la ejecución del analizador semántico, se devuelve el código generado al analizador sintáctico. Mediante “Executor”, cuyo formato se indica en la figura 27, este escribe el código a un archivo de texto .cs. Luego, “Executor” invoca al compilador C# de Microsoft, y el código almacenado en el archivo de texto se transforma en un archivo ejecutable. Automáticamente se ejecuta este archivo, mostrando al usuario el programa compilado. Si el compilador C# no puede compilar correctamente el código, se muestra el error dado, dando al usuario la posibilidad de corregirlo.

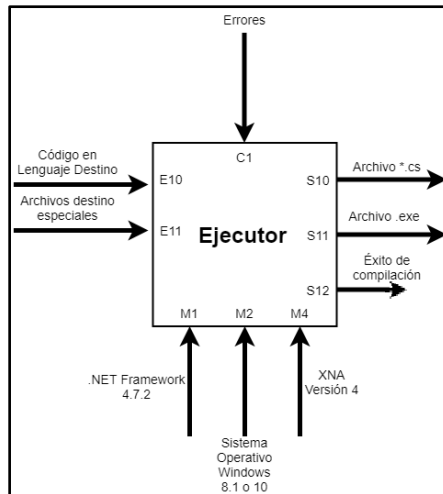


Fig. 27. Arquitectura del ejecutor del compilador.

2.10 – Pruebas

Se realizó una serie de pruebas para comprobar el funcionamiento de cada aspecto del compilador. Para el **análisis léxico**, se hizo pruebas con diferentes tipos de entradas de texto con el propósito de producir errores léxicos. Se intentó con varias cadenas de entrada: con números incompletos (como 0.), símbolos iniciales inválidos (#, @, etcétera). Las pruebas realizadas permitieron arreglar el código para eliminar cualquier error fatal que se podría dar durante la fase de exploración.

En el caso del **reconocedor**, fue necesario hacer pruebas con diferentes cantidades de código, tratando de cumplir cada regla gramatical en algún momento o incluso tratar de romper algunas reglas para ver el proceder del analizador. Los resultados obtenidos permitieron arreglar algunos errores (por ejemplo, errores causados por líneas vacías, etc.).

Las mismas pruebas realizadas para el reconocedor fueron de ayuda en la fase de análisis semántico, facilitando el arreglo de pequeñas deficiencias en las rutinas semánticas.

Viendo los resultados, errores y quejas de los primeros usuarios del compilador, se pudo corregir algunos errores, mejorar unas características y enviar los cambios mediante nuevas actualizaciones.

2.11 – Recopilación de Código Compilado

Al inicio, para la recopilación de datos acerca del uso del compilador, cada compilación invocaba un llamado a un hilo aparte que permitía enviar el código escrito al correo del desarrollador. Pero, debido a muchas políticas de seguridad a nivel de los proveedores de servicio de correo, fue necesario implementar una redundancia de correos; es decir que, si fallaba al enviar por una cuenta, se procedía a enviar por otra.

Se llegó a la conclusión de una mejor implementación: de armar una base de datos SQL en Azure y que el programa envíe directamente el código allí. De esta forma, sería más fácil bajar código ingresado, observar qué errores hay, cuántos programas compilaron exitosamente, entre otros datos. Al arrancar el programa, se obtiene el ID del usuario, que se utiliza para posteriores inserciones de código en la base de datos. Si es un usuario nuevo, se procede a crearlo. A continuación en la figura 28, se puede visualizar las tablas sencillas que fueron usadas para la base de datos:

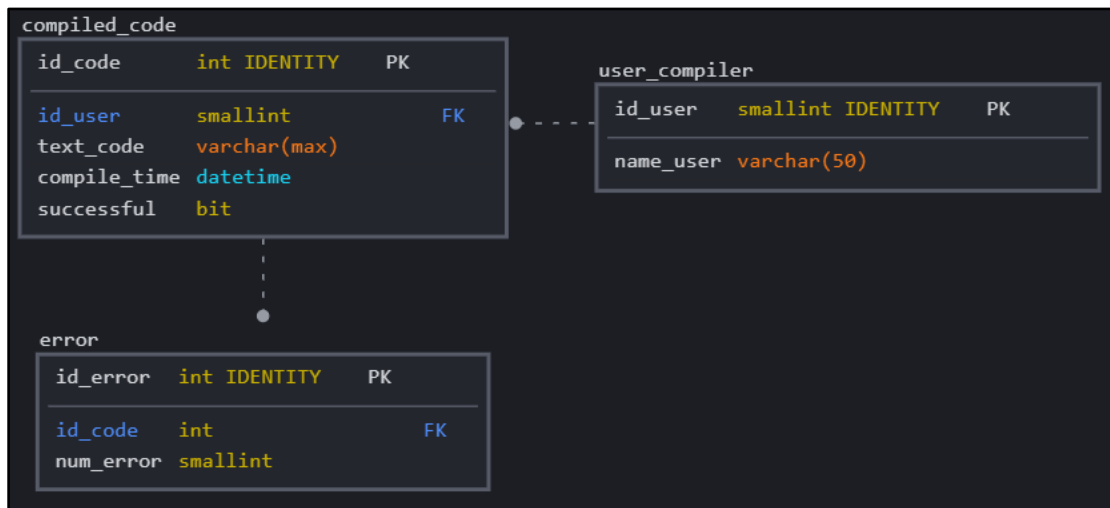


Fig. 28. Tablas de la base de datos usada para registrar usuarios, código compilado y errores detectados.

Aunque haya errores o el programa del usuario no se pueda compilar correctamente, el código se envía para poder hacer análisis sobre las fallas detectadas en el programa del usuario y poder realizar análisis estadísticos sobre ello. El funcionamiento de la base de datos se indica en la figura 29.

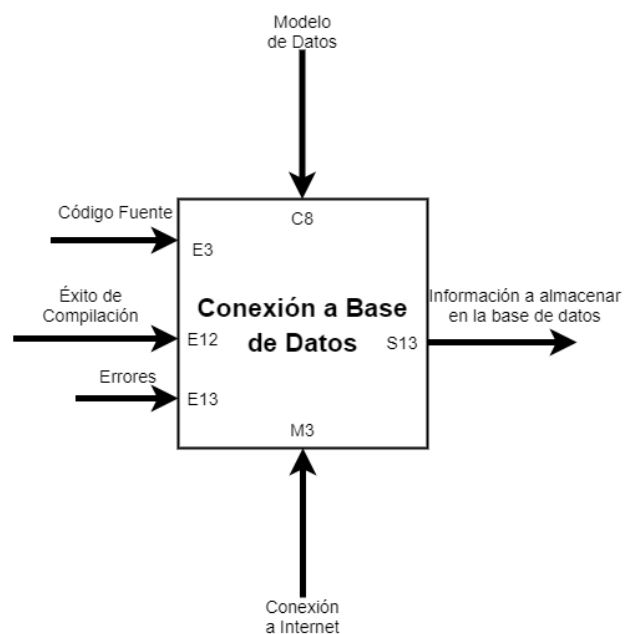


Fig. 29. Arquitectura de la conexión a la base de datos.

Se distribuyó un enlace a Dropbox donde se encontraba el instalador de la versión 1.0.0.0 del compilador. Además, se incluyó en el archivo comprimido los instaladores XNA requeridos para las funcionalidades gráficas y un manual de instalación. El enlace fue compartido con estudiantes de bachillerato y educación superior, para una fase de prueba y uso de parte de los usuarios, y una encuesta posterior.

2.12 – Método de Actualización Automática

Un sistema de actualización automática fue implementado permitiendo distribuir el programa, realizar pruebas e ir puliendo y arreglando los errores existentes y agregar nuevas funcionalidades.

Cada vez que el programa se ejecuta, busca en línea en un archivo .xml, alojado en un servidor de 000webhost.com. Se hace una comparación entre la versión actual del programa en ejecución y la versión descrita en el archivo .xml. Si el número de la versión del archivo en línea es más actual que la versión actualmente corriendo, se procede a descargar el archivo del ejecutable actualizado, almacenado en Dropbox. Para comprobar que el archivo ejecutable se transmitió con éxito, se compara el hash MD5 del archivo descargado con el hash registrado en el .xml. Si hay alguna discrepancia, se cierra el programa y se ejecuta de nuevo para poder tratar de actualizarse otra vez. Si los hashes tienen el mismo valor, el programa en ejecución se cierra y se ejecuta la nueva versión, reemplazando la antigua.

El usuario puede saber en cualquier momento qué versión del programa tiene haciendo clic en “Ayuda->Acerca de...”.

CAPÍTULO 3

Resultados

3.1 – Introducción a los Resultados

3.1.1 – Alcance

Para comprobar el cumplimiento de los objetivos planteados al inicio del proyecto, se evaluará dos características del ISO 25010:2011: la eficiencia de desempeño y la adecuación funcional.

La eficiencia de desempeño es una comparación del rendimiento con los recursos requeridos en un momento dado bajo las condiciones dadas (International Organization for Standardization, 2017). Las subcaracterísticas a tomar en cuenta son el comportamiento temporal y utilización de recursos.

La adecuación funcional representa el grado en que el programa provee las funciones que cumplen las necesidades bajo diferentes condiciones (International Organization for Standardization, 2017). Sus subcaracterísticas son la completitud, corrección y pertinencia funcional.

Con la intención de medir estas características y también obtener una serie de medidas cuantitativas diferentes, algunos usuarios colegiales y universitarios probaron el programa. Tras un periodo de prueba, los usuarios llenaron una encuesta para dar retroalimentación sobre varios aspectos del compilador.

3.1.2 – Categoría de Usuarios

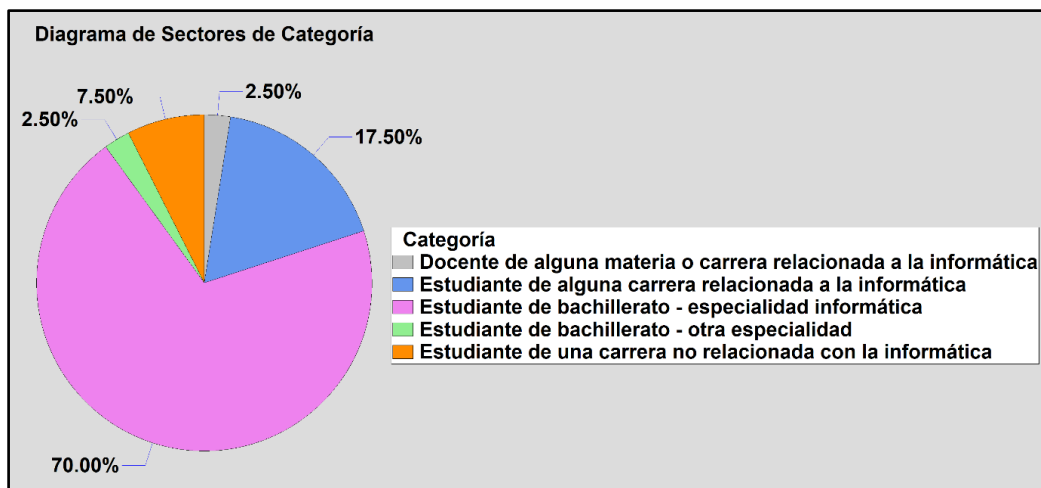


Fig. 30. Diagrama porcentual de tipos de usuarios que llenaron la encuesta.

La mayoría de los usuarios fueron estudiantes de bachillerato en la especialidad de informática, como se puede ver en la figura 30. La figura 31 muestra cuántos usuarios (de los que llenaron la encuesta) habían programado antes.



Fig. 31. Diagrama de usuarios que han programado antes de la utilización del compilador.

De los participantes, el 55% tienen experiencia en la programación y 45% no han programado antes de usar el compilador.

3.1.3 – Datos de la Encuesta

La encuesta que llenaron los usuarios tuvo un total de 24 preguntas. Debido al cambio del flujo de las preguntas de acuerdo con las respuestas y su obligatoriedad, se puede contestar un mínimo de 17 preguntas hasta un máximo de 22. Las únicas preguntas abiertas incluyen el nombre (obligatorio) y comentarios finales sobre lo que se podría mejorar (no obligatorio).

En promedio, los participantes se demoraron once minutos y medio en completar la encuesta, como indica la tabla 3.1. Este dato es muy alto debido a un usuario que se demoró más de tres horas en llenarla. El 75% de los usuarios demoraron 8 minutos o menos.

TABLA 3.1 Datos estadísticos del tiempo que los usuarios se tomaron para llenar la encuesta.

Resumen Estadístico para Duracion		Percentiles para Duracion	
		Percentiles	
Recuento	40	1.00%	0:00:31
Promedio	0:11:30	5.00%	0:01:06
Desviación Estándar	0:29:49	10.00%	0:02:08
Coeficiente de Variación	259.42%	25.00%	0:03:31
Mínimo	0:00:31	50.00%	0:04:46
Máximo	3:10:43	75.00%	0:08:05
Rango	3:10:12	90.00%	0:20:04
Sesgo Estandarizado	15.1309	95.00%	0:28:31
Curtosis Estandarizada	46.2377	99.00%	3:10:43

3.1.4 – Descripción de Resultados

Para realizar análisis cuantitativos sobre preguntas de sí o no, se ha decidido cuantificar las respuestas negativas a un valor de cero y las positivas a 1. En caso de haber un valor intermedio (más o menos, tal vez), este se convierte en 0.5.

3.2 – Medida de la Eficiencia de Desempeño (ISO 25010:2011)

3.2.1 – Comportamiento Temporal

Una de las maneras empleadas para medir la eficiencia de desempeño del ISO 25010:2011 fue mediante el uso de un cronómetro en el código para tomar el tiempo en varios puntos del compilador para descubrir el tiempo promedio requerido, en milisegundos, para realizar las diferentes tareas de compilación. Específicamente, estas pruebas permiten medir la subcaracterística “comportamiento temporal”. Cada prueba fue realizada cinco veces con diferentes archivos de código fuente. Dos de las pruebas fueron con ventana gráfica XNA y tres con consola textual. La prueba fue realizada dentro del depurador de Visual Studio 2019 y también fuera de ello para comparar la diferencia de tiempo. Además, estas pruebas fueron realizadas en dos computadoras; una con procesador Intel Core i5 10ma generación (seis núcleos a 2.9ghz), 16gb de RAM DDR4 y disco SSD de interfaz M.2 con capacidad de 480gb (denominada Computadora “A”). La otra computadora usada tiene un procesador Intel Pentium (dos núcleos a 3ghz), 16gb de RAM DDR3 y un disco HDD con interfaz SATA 3 con capacidad de 500gb (denominada Computadora “B”). Ambos equipos llevan el sistema operativo Windows 10.

A continuación, en la tabla 3.2, se detallan los tiempos requeridos, en milisegundos, para el primer conjunto de pruebas realizadas en la Computadora “A”:

TABLA 3.2 Datos informativos de los tiempos de carga y procesamiento en la computadora de prueba A.

Computadora "A"	Con Depuración	Sin Depuración
Cargar Archivos Principales	183ms	143ms
Cargar Código Fuente	5.8ms	5.0ms
Analizar en búsqueda de errores	371ms	10ms
Preprocesamiento léxico	0.2ms	0.2ms
Primera compilación	393ms	10ms
Tiempo de escritura, segunda compilación y ejecución	911ms	897ms
Tiempo de escritura, segunda compilación y ejecución (-500ms)	411ms	397ms
Tiempo total de compilación, escritura y ejecución	1318ms	904ms

Ahora se detallarán en la tabla 3.3 los tiempos tomados para las pruebas realizadas en la Computadora "B":

TABLA 3.3 Datos informativos de los tiempos de carga y procesamiento en la computadora de prueba B.

Computadora "B"	Con Depuración	Sin Depuración
Cargar Archivos Principales	838ms	477ms
Cargar Código Fuente	25ms	21ms
Analizar en búsqueda de errores	33ms	27ms
Preprocesamiento léxico	2.0ms	1.8ms
Primera compilación	27ms	23ms
Tiempo de escritura, segunda compilación y ejecución	1804ms	1688ms
Tiempo de escritura, segunda compilación y ejecución (-500ms)	1304ms	1188ms
Tiempo total de compilación, escritura y ejecución	2537ms	1647ms

Como se puede observar, hay mucha diferencia entre el tiempo requerido durante ejecución entre computadora "A" y "B". En el caso de menor diferencia, el tiempo total de compilación varía por un factor de 1.8. Analizando el caso opuesto, donde hay mayor diferencia, se encuentra en el tiempo requerido para preprocesamiento léxico con un factor de 9. En la figura 32, se puede observar una comparación del tiempo requerido para ambas computadoras.

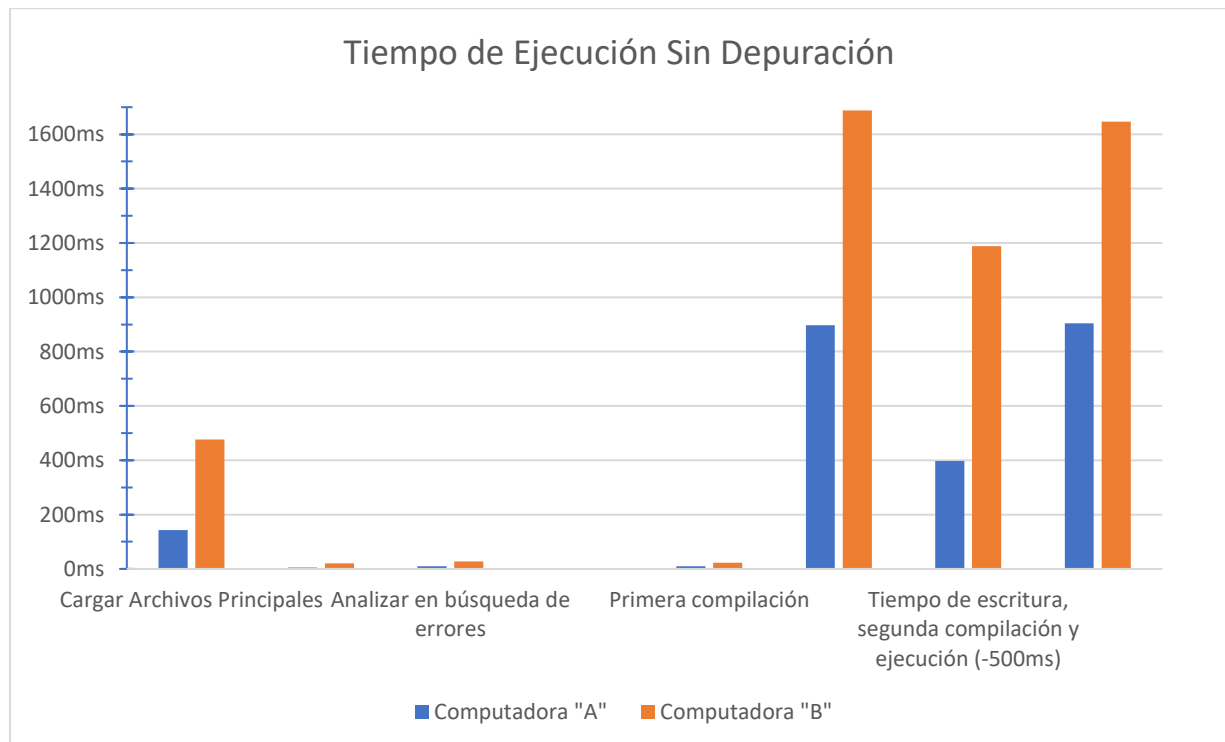


Fig. 32. Comparación entre respuesta de tiempo en ambas computadoras durante ejecución normal.

La computadora “A” compila con una velocidad mucho mayor que la computadora “B”, pero eso no quiere decir que el segundo caso representa una gran pérdida de tiempo; la diferencia total de compilación entre ambos equipos es menos de 750ms. Para un estudiante que esté recién aprendiendo a programar y que esté probando las diferentes funcionalidades del compilador y su sintaxis, no requerirá compilaciones tan seguidas ni tan rápidas. El promedio de tiempo total de compilación en la computadora “B” es menos de dos segundos, lo cual parece ser un tiempo aceptable.

De cuarenta usuarios que probaron el compilador o que al menos pudieron observarlo en ejecución, sacó un puntaje de 7.625 en cuanto al rendimiento percibido, como se puede apreciar en la tabla 3.4. En la escala usada, 1 representa una velocidad demasiada lenta, 5 es una rapidez suficiente y 10 muestra mucha velocidad. Analizando los cuartiles de esta pregunta de la encuesta, el primero tiene un valor de 5, mostrando que el rendimiento fue suficiente para casi todos los usuarios.

TABLA 3.4 Datos estadísticos del rendimiento del compilador según la encuesta realizada.

Resumen Estadístico para Rendimiento Percibido		Percentiles para Rendimiento Percibido	
		Percentiles	
Recuento	40	1.00%	1
Promedio	7.625	5.00%	3
Desviación Estándar	2.42516	10.00%	5
Coeficiente de Variación	31.81%	25.00%	5
Mínimo	1	50.00%	8
Máximo	10	75.00%	10
Rango	9	90.00%	10
Sesgo Estandarizado	-2.84771	95.00%	10
Curtosis Estandarizada	1.01458	99.00%	10

En la pregunta abierta y opcional al final de la encuesta, cinco de los usuarios específicamente pidieron mayor velocidad en el proceso de compilación. Un factor que puede haber bajado este puntaje es el hecho de que alrededor de 27 estudiantes que llenaron la encuesta vieron el compilador a través de una clase virtual. El equipo usado para mostrar el uso del compilador fue de bajo rendimiento.

3.2.2 – Utilización de Recursos - RAM

Otra subcaracterística que se analizará sobre la eficiencia de desempeño es la utilización de recursos. La cantidad de RAM requerida, en MBs, durante varias fases de operación del compilador se muestra en la tabla 3.5.

TABLA 3.5 Datos informativos sobre la utilización de memoria RAM para diferentes etapas de funcionamiento del compilador.

Fases	Con Depuración	Sin Depuración
Pantalla de bienvenida	83mb	53mb
Antes de mostrar interfaz	38mb	16mb
Interfaz cargada	40mb	19mb
Cargando código fuente 1	42mb	21mb
Cargando código fuente 2	42mb	22mb
Cargando código fuente 3	42mb	22mb
Análisis de código fuente	44mb	24mb
Compilación de programa	44mb	52mb
Cuadro de diálogo para guardar	51mb	28mb
Cuadro de diálogo para abrir	57mb	30mb

Como se muestra, el total de memoria requerida durante una ejecución normal no llega a una gran cantidad; solo 53mb en el caso máximo, con un promedio de 29mb. Para una computadora que tenga una cantidad mínima de memoria RAM (de al menos 3gb), no representará una gran pérdida de recursos el uso de 22mb a 53mb.

3.2.3 – Utilización de Recursos - Procesador

Continuando con la subcaracterística de la utilización de recursos, se decidió también hacer un análisis de cuánto procesamiento es requerido para usar el compilador. Se realizó la prueba en Computadora “A” y “B”, como en la prueba de tiempo de respuesta, mostrado en la tabla 3.6.

TABLA 3.6 Datos comparativos de la utilización de recursos del procesador en las computadoras de prueba.

Computadora "A"	Con Depuración	Sin Depuración	Computadora "B"	Con Depuración	Sin Depuración
Pantalla de Bienvenida	7%	1.70%	Pantalla de Bienvenida	43%	9.40%
Cargando código fuente 1	2%	0.40%	Cargando código fuente 1	17%	2.40%
Cargando código fuente 2	3%	0.40%	Cargando código fuente 2	6%	2.50%
Cargando código fuente 3	3%	0.20%	Cargando código fuente 3	6%	2.60%
Primer análisis de código fuente	8%	1%	Primer análisis de código fuente	25%	4.10%
Primera compilación	8%	5.70%	Primera compilación	22%	18%
Posteriores análisis Posteriores compilaciones	3%	1.00%	Posteriores análisis Posteriores compilaciones	10%	1.00%
Cuadro de diálogo para abrir	22%	2.90%	Cuadro de diálogo para abrir	71%	21.90%
Cuadro de diálogo para guardar	12%	2.90%	Cuadro de diálogo para guardar	71%	24%

Es aparente que la segunda computadora requiere mayor uso de su procesador debido a las mayores limitaciones que este tiene en velocidad de reloj y cantidad de núcleos. Las instancias que requieren mayor uso del procesador son momentos efímeros; para la mayoría de la ejecución, se requiere menos de 3% de utilización de los ciclos de reloj del procesador cuando el compilador esté realizando la mayor parte de su trabajo.

Según los análisis de utilización de recursos, el compilador puede funcionar en computadoras de bajo rendimiento, aunque se sugiere no sobrecargarlas con otros programas corriendo simultáneamente.

3.3 – Medida de la Adecuación Funcional (ISO 25010:2011)

3.3.1 – Completitud Funcional

La completitud funcional es “el grado en que un conjunto de funciones cubre todas las tareas especificadas y objetivos de usuario” (International Organization for Standardization, 2017). Aunque no se ha especificado explícitamente las tareas requeridas por los usuarios, se podría establecer las siguientes:

1. Guardar y abrir archivos de código fuente.
2. Ingresar código con palabras reservadas en español, conforme a una lista de reglas lexicales, sintácticas y semánticas, que se transforme correctamente en un programa ejecutable.
3. Ingresar código que permita teclear letras especiales españolas (é, ñ, ü, etc.).
4. Observar en tiempo real la ubicación de errores y una descripción sobre cómo corregirlos.
5. Ingresar código para diferentes estructuras de una forma rápida.
6. Tener la opción de mostrar los diferentes valores de las variables mediante una prueba de escritorio.
7. Permitir la personalización de tipo, tamaño y color de letra y la interfaz.
8. Recuperación automática de errores.
9. Creación de programas textuales (consola) y gráficas (ventana XNA) con otros aspectos multimediales incorporados (sonidos, música y colores).
10. Incorporación de materiales didácticos que enseñen el uso del programa (manuales, ejercicios y ejemplos).

El grado de completitud de las tareas previamente explicitadas se detalla a continuación:

1. 100% - Se permite abrir y guardar archivos de código fuente, guardar como, crear nuevos e incluso abrir ejemplos de código, modificarlos y volver a guardarlos aparte.

2. 100% - Las palabras reservadas permitidas por el lenguaje son españolas. Se acepta un código que conforma a reglas explícitas, y el código ingresado se transforma a un archivo ejecutable.
3. 100% - Se puede ingresar variables, declarar funciones y escribir palabras reservadas que utilicen letras españolas especiales normalmente prohibidas en otros lenguajes.
4. 50% - El código es analizado cada tres segundos (si el usuario lo desea) y se indica la ubicación aproximada del error. El mensaje no es siempre lo suficientemente descriptivo para corregir el error, y la ubicación no es siempre la correcta. Además, no se permite la visualización en el código de la ubicación de errores sintácticos ni semánticos.
5. 100% - Mediante clic derecho, se puede insertar una plantilla de código de estructuras de control y bucles, mitigando la cantidad de errores sintácticos de parte del usuario.
6. 75% - Existe la opción de activar pruebas de escritorio y seleccionar las variables a monitorear. La desventaja de este proceso es que no se realiza en tiempo real, sino al final de la ejecución del código.
7. 0% - Aún no se ha implementado opciones de personalización del código fuente.
8. 50% - Algunos errores son recuperables. Los errores lexicales, en su mayoría, se pueden recuperar. Algunos errores sintácticos y misceláneos se pueden recuperar automáticamente también. Se podría mejorar el proceso de recuperación de errores semánticos y sintácticos.
9. 90% - Se puede crear fácilmente programas de consola que permitan el manejo de sonidos y música. No se permite cambiar los colores del texto de la consola. También hay la posibilidad de desarrollar programas gráficos que usen XNA Framework. Se permite mostrar gráficos en la pantalla y cambiar sus colores.
10. 75% - Existen manuales, ejercicios y ejemplos incorporados en el compilador. Los manuales podrían especificar más detalles del compilador y el lenguaje. Los ejemplos no representan la totalidad de las funcionalidades incorporadas, pero sí dan una buena idea de lo que se puede hacer. Por último, los ejercicios deberían tener un mejor formato e ir aumentando de dificultad.

Hay un cumplimiento de aproximadamente 74% de la completitud funcional, basándose en las funciones requeridas detalladas.

Dos preguntas de la encuesta podrían encajarse en esta subcaracterística: “¿Usted cree que este compilador se podría usar como un primer lenguaje de programación en los colegios?” y “Califique qué tanto fueron cumplidas sus expectativas al usar el compilador.” Los resultados de ambas preguntas se pueden ver en las tablas 3.7 y 3.8.

TABLA 3.7 Datos estadísticos sobre el criterio de los usuarios sobre poder usar el compilador como primer lenguaje de programación.

Resumen Estadístico para Primer Lenguaje		Percentiles para Primer Lenguaje	
		Percentiles	
Recuento	40	1.00%	0
Promedio	0.825	5.00%	0.5
Desviación Estándar	0.266747	10.00%	0.5
Coeficiente de Variación	32.33%	25.00%	0.5
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-3.04293	95.00%	1
Curtosis Estandarizada	0.557334	99.00%	1

Más de los dos tercios de los usuarios creen que se podría usar el Compilador Córdor y su respectivo lenguaje como un primer lenguaje de programación en los colegios. Se podría asignar un puntaje de 82.5% de completitud funcional basándose en esta pregunta, ya que el compilador fue creado con intenciones de servir como un primer lenguaje de aprendizaje.

TABLA 3.8 Datos estadísticos sobre el cumplimiento de las expectativas del compilador a los usuarios.

Resumen Estadístico para Cumplimiento Expectativas		Percentiles para Cumplimiento Expectativas	
		Percentiles	
Recuento	40	1.00%	5
Promedio	8.3	5.00%	5
Desviación Estándar	1.87014	10.00%	5
Coeficiente de Variación	22.53%	25.00%	7
Mínimo	5	50.00%	9
Máximo	10	75.00%	10
Rango	5	90.00%	10
Sesgo Estandarizado	-1.76698	95.00%	10
Curtosis Estandarizada	-1.26983	99.00%	10

Esta pregunta muestra un grado de 83% de satisfacción de los usuarios en cuanto al cumplimiento de expectativas del compilador. Analizando los cuartiles, son 7, 9 y 10 respectivamente, mostrando una gran tendencia hacia una satisfacción completa o casi completa.

En base a los tres valores posibles de completitud funcional, se podría calcular un promedio de **79.8%** en esta subcaracterística.

3.3.2 – Corrección Funcional

Esta subcaracterística representa “el grado en que un producto o sistema provee los resultados correctos con el grado requerido de precisión” (International Organization for

Standardization, 2017). Es difícil tener un método objetivo para medir esta característica ya que existe posibilidades y combinaciones infinitas dentro del lenguaje. Incluso para redactar una sola instrucción, hay incontables mezclas posibles de caracteres y lexemas para producir código legítimo con la funcionalidad correcta implicada.

Se puede decir que, mediante un análisis subjetivo, la corrección funcional es mayor al 95%.

Entre las últimas preguntas de la encuesta, hay una pregunta que pide una calificación de la exactitud percibida del código compilado. Se especifica la palabra “percibida” porque puede que haya usuarios que compilaron código pensando que tendría un resultado específico, cuando en realidad debía tener otro. Los resultados de la pregunta se muestran en la tabla 3.9.

TABLA 3.9 Datos estadísticos sobre la exactitud del compilador a criterio de los usuarios.

Resumen Estadístico para Exactitud Percibida		Percentiles para Exactitud Percibida	
		Percentiles	
Recuento	40	1.00%	1
Promedio	8.15	5.00%	2.5
Desviación Estándar	2.45524	10.00%	5
Coefficiente de Variación	30.13%	25.00%	7
Mínimo	1	50.00%	9
Máximo	10	75.00%	10
Rango	9	90.00%	10
Sesgo Estandarizado	-3.6629	95.00%	10
Curtosis Estandarizada	1.6106	99.00%	10

Los usuarios dieron un puntaje promedio de 81.5% en esta pregunta acerca de la corrección del código compilado. Los cuartiles muestran una tendencia hacia la perfección de la exactitud; 7, 9 y 10 respectivamente.

Aunque los dos valores analizados para la corrección funcional podrían ser más altos, su promedio se tomará en cuenta para la evaluación de esta subcaracterística. Entre el valor inicial calculado y la calificación de los usuarios, el compilador tiene aproximadamente **88.25%** de corrección funcional.

3.3.3 – Pertinencia Funcional

Este aspecto se puede medir, mostrando “el grado en que las funciones facilitan el cumplimiento de tareas y objetivos especificados” (International Organization for Standardization, 2017). La medición de esta subcaracterística será mediante el análisis de seis preguntas de la encuesta.

1. *¿Usted cree que el haber programado con palabras reservadas en español fue más fácil que en inglés?*

Esta pregunta se enlaza directamente con la función 2 de la lista de funciones en la sección 3.3.1, y los resultados se muestran en la tabla 3.10.

TABLA 3.10 Datos estadísticos sobre la facilidad de las palabras reservadas del compilador según los usuarios.

Resumen Estadístico para Palabras Reservadas Más Fáciles		Percentiles para Palabras Reservadas Más Fáciles	
		Percentiles	
Recuento	22	1.00%	0
Promedio	0.704545	5.00%	0
Desviación Estándar	0.398183	10.00%	0
Coefficiente de Variación	56.52%	25.00%	0.5
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-1.75782	95.00%	1
Curtosis Estandarizada	-0.692758	99.00%	1

Esta pregunta muestra que, de los 22 usuarios que tienen experiencia programando, hay una aceptación aproximada de 70.5% que programar con palabras reservadas en español es más fácil que en inglés.

2. *¿Usted cree que la sintaxis del lenguaje fue más fácil o intuitivo que la sintaxis de otros lenguajes?*

Este interrogante representa también la función 2, pero se centra más en la gramática reconocida por el lenguaje. Los resultados se ven en la tabla 3.11.

TABLA 3.11 Datos estadísticos sobre la facilidad de la sintaxis según el criterio de los usuarios del compilador.

Resumen Estadístico para Sintaxis Fácil		Percentiles para Sintaxis Fácil	
		Percentiles	
Recuento	22	1.00%	0
Promedio	0.931818	5.00%	0.5
Desviación Estándar	0.233781	10.00%	1
Coefficiente de Variación	25.09%	25.00%	1
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-6.93379	95.00%	1
Curtosis Estandarizada	12.7049	99.00%	1

Los participantes dieron un puntaje promedio de 93% en cuanto a la facilidad de la sintaxis de este lenguaje frente a otros. Más del 90% de los usuarios están de acuerdo en este aspecto.

3. *Califique qué tan efectivos fueron los manuales y ejercicios en enseñarle el lenguaje del compilador.*

La décima funcionalidad es representada en esta pregunta.

TABLA 3.12 Datos estadísticos sobre la efectividad de los manuales proporcionados a los usuarios del compilador.

Resumen Estadístico para Efectividad Manuales		Percentiles para Efectividad Manuales	
		Percentiles	
Recuento	32	1.00%	5
Promedio	8.78125	5.00%	5
Desviación Estándar	1.53947	10.00%	7
Coefficiente de Variación	17.53%	25.00%	8
Mínimo	5	50.00%	9
Máximo	10	75.00%	10
Rango	5	90.00%	10
Sesgo Estandarizado	-3.2768	95.00%	10
Curtosis Estandarizada	1.53606	99.00%	10

De los 32 usuarios que sí revisaron los manuales y ejercicios incorporados en el compilador, dieron un puntaje promedio de 87.8% en su efectividad. Considerando que el puntaje mínimo es 5 y que los cuartiles son 8, 9 y 10 respectivamente, los manuales fueron relevantes y útiles en el aprendizaje.

4. *¿Considera usted que la implementación de diferentes colores en el código fuente fue efectiva?*

Aunque esta pregunta no tiene un vínculo directo con ninguna de las funciones, esta es una característica general que ayuda en el uso del compilador.

TABLA 3.13 Datos estadísticos sobre la efectividad de los colores usados para el código fuente.

Resumen Estadístico para Efectividad Colores		Percentiles para Efectividad Colores	
		Percentiles	
Recuento	40	1.00%	0
Promedio	0.925	5.00%	0.5
Desviación Estándar	0.213337	10.00%	0.5
Coefficiente de Variación	23.06%	25.00%	1
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-7.77848	95.00%	1
Curtosis Estandarizada	11.9098	99.00%	1

La gran mayoría de los participantes concordaron en la efectividad de los colores implementados en el código. El puntaje promedio es 92.5%. Todos los cuartiles son positivos en la efectividad calificada de los colores de parte de los usuarios.

5. *¿Qué tan efectiva fue la tabla de errores en ayudarlo a eliminar errores?*

Esta pregunta se enlaza con la función 4. La tabla de errores es importante en la erradicación de falencias en el código y permite corregirlo.

TABLA 3.14 Datos estadísticos sobre la efectividad de la tabla de errores proporcionada en el compilador según los usuarios.

Resumen Estadístico para Efectividad Tabla de Errores		Percentiles para Efectividad Tabla de Errores	
		Percentiles	
Recuento	40	1.00%	1
Promedio	3.875	5.00%	2.5
Desviación Estándar	0.991955	10.00%	3
Coeficiente de Variación	25.60%	25.00%	3
Mínimo	1	50.00%	4
Máximo	5	75.00%	5
Rango	4	90.00%	5
Sesgo Estandarizado	-1.46591	95.00%	5
Curtosis Estandarizada	0.162175	99.00%	5

A diferencia de muchas de las otras preguntas, esta tuvo opciones posibles entre 1 y 5 para la calificación de la efectividad de la tabla de errores. Al multiplicar el puntaje promedio por dos, se obtendría 77.5%. Los cuartiles son 3,4 y 5. Esta pregunta no tiene una calificación tan alta como la mayoría de las otras.

6. *Califique qué tan útil le fue la inserción rápida de código.*

La función 5 es representada con esta pregunta. Esto ayuda en la reducción de errores de sintaxis para nuevos aprendices y agiliza la escritura de código para todos.

TABLA 3.15 Datos estadísticos sobre la efectividad de la opción de inserción rápida de bucles y estructuras de control.

Resumen Estadístico para Efectividad Inserción Rápida		Percentiles para Efectividad Inserción Rápida	
		Percentiles	
Recuento	24	1.00%	5
Promedio	8.75	5.00%	5
Desviación Estándar	1.62186	10.00%	7
Coeficiente de Variación	18.54%	25.00%	7.5
Mínimo	5	50.00%	9.5
Máximo	10	75.00%	10
Rango	5	90.00%	10
Sesgo Estandarizado	-2.31806	95.00%	10
Curtosis Estandarizada	0.343075	99.00%	10

Los usuarios calificaron la inserción de código con un puntaje promedio de 87.5%. Los cuartiles de 7.5, 9.5 y 10 muestran que pocos usuarios no encontraron la inserción rápida efectiva.

Haciendo un promedio de los puntajes de las seis preguntas anteriores, se obtiene 84.82% en la pertinencia funcional.

3.4 – Análisis de Datos Adicionales

3.4.1 – Encuesta

- *Califique su primera impresión de la programación al usar el compilador.*

TABLA 3.16 Datos estadísticos sobre la primera impresión de los usuarios acerca de la programación.

Resumen Estadístico para Primera Impresión		Percentiles para Primera Impresión	
		Percentiles	
Recuento	18	1.00%	1
Promedio	7.55556	5.00%	1
Desviación Estándar	2.85373	10.00%	3
Coefficiente de Variación	37.77%	25.00%	5
Mínimo	1	50.00%	9
Máximo	10	75.00%	10
Rango	9	90.00%	10
Sesgo Estandarizado	-1.63658	95.00%	10
Curtosis Estandarizada	-0.16081	99.00%	10

Los usuarios nuevos en la programación han dado una calificación de 75.56% en cuanto a su impresión de la materia al usar el compilador, como se puede ver en la tabla 3.16.

- *¿Usted cree que aprender con el Compilador Cóndor le impulsará a seguir estudiando la programación o alguna carrera relacionada a la programación?*

TABLA 3.17 Datos sobre el interés de los usuarios para continuar con una carrera relacionada con la programación.

Resumen Estadístico para Seguir Estudiando		Percentiles para Seguir Estudiando	
		Percentiles	
Recuento	18	1.00%	0
Promedio	0.805556	5.00%	0
Desviación Estándar	0.303842	10.00%	0.5
Coefficiente de Variación	37.72%	25.00%	0.5
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-2.35839	95.00%	1
Curtosis Estandarizada	0.975423	99.00%	1

De los estudiantes nuevos en la programación hay un puntaje de 80.56% con respecto a las posibilidades de seguir estudiando una carrera relacionada a la informática, gracias al uso del compilador.

- *¿Usted cree que programar con palabras reservadas en español sería más fácil para nuevos estudiantes de programación?*

TABLA 3.18 Datos estadísticos sobre la facilidad de la programación con palabras reservadas en español.

Resumen Estadístico para Palabras Reservadas Nuevos		Percentiles para Palabras Reservadas Nuevos	
		Percentiles	
Recuento	22	1.00%	0
Promedio	0.886364	5.00%	0
Desviación Estándar	0.305965	10.00%	0.5
Coeficiente de Variación	34.52%	25.00%	1
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-4.9814	95.00%	1
Curtosis Estandarizada	5.39446	99.00%	1

Esta pregunta fue realizada a los usuarios que han programado antes de usar el compilador. Entre los usuarios que contestaron esta pregunta, hay un promedio de 88.64% de creencia que programar con palabras reservadas en español es más fácil para nuevos estudiantes. Todos los cuartiles indican una respuesta positiva en esta pregunta.

- *¿Usted cree que programar con la sintaxis del lenguaje sería más fácil para nuevos estudiantes de programación?*

TABLA 3.19 Datos estadísticos de la facilidad del sintaxis del lenguaje para nuevos estudiantes.

Resumen Estadístico para Sintaxis Nuevos Estudiantes		Percentiles para Sintaxis Nuevos Estudiantes	
		Percentiles	
Recuento	22	1.00%	0
Promedio	0.818182	5.00%	0
Desviación Estándar	0.328976	10.00%	0.5
Coeficiente de Variación	40.21%	25.00%	0.5
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-3.17808	95.00%	1
Curtosis Estandarizada	1.61521	99.00%	1

Entre los usuarios que contestaron esta pregunta, hay 81.82% de aceptación en la facilidad de la sintaxis del lenguaje para que lo aprendan nuevos estudiantes.

- *Califique su experiencia/impresión del compilador.*

TABLA 3.20 Datos estadísticos sobre la experiencia de los usuarios del compilador durante su utilización.

Resumen Estadístico para Experiencia		Percentiles para Experiencia	
		Percentiles	
Recuento	40	1.00%	1
Promedio	8.25	5.00%	5
Desviación Estándar	2.12132	10.00%	5
Coeficiente de Variación	25.71%	25.00%	7
Mínimo	1	50.00%	9
Máximo	10	75.00%	10
Rango	9	90.00%	10
Sesgo Estandarizado	-3.47682	95.00%	10
Curtosis Estandarizada	2.53778	99.00%	10

Con la excepción de unas respuestas bajas, la mayoría de las personas han calificado con un buen puntaje su experiencia del compilador. El puntaje promedio es 82.5%. Analizando los cuartiles, se puede observar los valores de 7, 9 y 10, los que apuntan a que la mayoría de los usuarios estuvieron muy satisfechos con el uso del compilador.

- *¿A usted le interesó o le gustó el manejo de características multimediales en el compilador?*

TABLA 3.21 Datos estadísticos sobre los criterios de las características de multimedia que ofrece el compilador a los usuarios.

Resumen Estadístico para Características Multimediales		Percentiles para Características Multimediales	
		Percentiles	
Recuento	40	1.00%	0.5
Promedio	0.925	5.00%	0.5
Desviación Estándar	0.18081	10.00%	0.5
Coeficiente de Variación	19.55%	25.00%	1
Mínimo	0.5	50.00%	1
Máximo	1	75.00%	1
Rango	0.5	90.00%	1
Sesgo Estandarizado	-5.26109	95.00%	1
Curtosis Estandarizada	2.92097	99.00%	1

La mayoría de los participantes acordaron en el interés despertado por las funcionalidades multimediales. De hecho, no hubo ninguna respuesta negativa al contestar esta pregunta. Los usuarios llegaron a tener un 92.5% de aceptación. A pesar de este alto valor, podría ser más alto ya que algunos de los usuarios que contestaron “más o menos” en esta pregunta no vieron ni probaron los aspectos multimediales.

- *¿Usted cree que la implementación de características multimediales podría ayudar a aumentar el interés de aprendizaje en nuevos estudiantes de programación?*

TABLA 3.22 Datos estadísticos del criterio de los usuarios sobre si la posibilidad de que las características multimedia generaren un mayor interés a nuevos estudiantes de carreras relacionadas con la programación.

Resumen Estadístico para Interés Multimedia		Percentiles para Interés Multimedia	
		Percentiles	
Recuento	40	1.00%	0
Promedio	0.95	5.00%	0.5
Desviación Estándar	0.189466	10.00%	1
Coeficiente de Variación	19.94%	25.00%	1
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-10.605	95.00%	1
Curtosis Estandarizada	22.6799	99.00%	1

Los participantes tienen 95% de aceptación en que los aspectos multimediales del compilador podrán servir para aumentar el interés en nuevos estudiantes para programar. Enseñar la programación a estudiantes de colegio no debería tener solo aspectos técnicos sino también se debe incluir diferentes características que aumenten el interés en ella.

- *¿Usted recomendaría este compilador a alguien que recién esté comenzando a programar?*

TABLA 3.23 Datos estadísticos sobre si el compilador es recomendable para los nuevos usuarios.

Resumen Estadístico para Recomendación		Percentiles para Recomendación	
		Percentiles	
Recuento	40	1.00%	0
Promedio	0.925	5.00%	0.5
Desviación Estándar	0.213337	10.00%	0.5
Coeficiente de Variación	23.06%	25.00%	1
Mínimo	0	50.00%	1
Máximo	1	75.00%	1
Rango	1	90.00%	1
Sesgo Estandarizado	-7.77848	95.00%	1
Curtosis Estandarizada	11.9098	99.00%	1

Esta última pregunta representa un buen cierre a la encuesta, con un promedio de 92.5% de aceptación en recomendar el compilador a alguien que esté aprendiendo.

3.4.2 – Compilaciones de Código

Durante las pruebas del compilador, se registró un total de 48 usuarios, 4074 compilaciones y 2661 errores detectados por el compilador. Algunos de los usuarios eran de

prueba o fueron creados accidentalmente; al eliminarlos, quedan 42 usuarios. Quitando las compilaciones y errores asociados a los usuarios eliminados, se cuenta con 3854 compilaciones reales y 2633 errores detectados.

Al analizar las compilaciones realizadas por cada usuario, se puede obtener los siguientes datos:

TABLA 3.24 Datos estadísticos de la cantidad de compilaciones de los usuarios del compilador.

Resumen Estadístico para Compilaciones por Usuario		Percentiles para Compilaciones por Usuario	
Recuento	42	Percentiles	
Promedio	91.7619	1.00%	0
Desviación Estándar	109.445	5.00%	0
Coefficiente de Variación	119.27%	10.00%	3
Mínimo	0	25.00%	24
Máximo	493	50.00%	52.5
Rango	493	75.00%	125
Sesgo Estandarizado	6.19861	90.00%	204
Curtosis Estandarizada	8.52332	95.00%	211
		99.00%	493

En promedio, cada usuario realizó 92 compilaciones. Según los cuartiles, tenemos 24, 52.5 y 125. Suponiendo que debe haber más de diez compilaciones para considerar que el usuario haya experimentado y probado bien al compilador, hay 33 personas que experimentaron a fondo el compilador.

A continuación, se analizará las equivocaciones cometidas por cada usuario.

TABLA 3.25 Datos estadísticos sobre la cantidad de equivocaciones de los usuarios al compilar sus códigos.

Resumen Estadístico para Porcentaje Errores		Percentiles para Porcentaje Errores	
Recuento	33	Percentiles	
Promedio	0.500617	1.00%	9.52%
Desviación Estándar	0.212074	5.00%	11.63%
Coefficiente de Variación	42.36%	10.00%	26.40%
Mínimo	0.0952381	25.00%	36.27%
Máximo	1	50.00%	47.06%
Rango	0.904762	75.00%	64.81%
Sesgo Estandarizado	0.708454	90.00%	72.22%
Curtosis Estandarizada	0.0263775	95.00%	93.88%
		99.00%	100.00%

En promedio, los usuarios tuvieron 50% de errores detectados al momento de compilar. Muchos de los usuarios nunca habían programado y solamente tuvieron una clase corta acerca del uso del compilador; esto podría haber contribuido a esta alta cantidad de errores.

Además, algunas de estas compilaciones no fueron por errores detectados en el código, sino por algún otro problema, como archivos en uso o ejecución.

Analizando la cantidad de errores detectados en cada compilación, se puede descubrir que, en promedio, cada compilación lleva 0.68 errores. En el máximo caso, se llegó a detectar diez errores en una sola compilación. El segundo cuartil tiene un valor de cero; esto nos quiere decir que más de la mitad de las compilaciones no tuvieron ningún error, y algunos casos extremos sí tuvieron varios.

TABLA 3.26 Datos estadísticos sobre la cantidad de errores por compilación de los usuarios.

Resumen Estadístico para Errores por Compilación		Percentiles para Errores por Compilación	
		Percentiles	
Recuento	3854	1.00%	0
Promedio	0.683186	5.00%	0
Desviación Estándar	0.975454	10.00%	0
Coefficiente de Variación	142.78%	25.00%	0
Mínimo	0	50.00%	0
Máximo	10	75.00%	1
Rango	10	90.00%	2
Sesgo Estandarizado	61.1216	95.00%	2
Curtosis Estandarizada	134.927	99.00%	4

TABLA 3.27 Tabla de frecuencia sobre la cantidad de errores detectados en cada compilación.

Tabla de Frecuencia para Errores por Compilación					
Clase	Valor	Frecuencia	Frecuencia Relativa	Frecuencia Acumulada	Frecuencia Rel. acum.
1	0	2072	0.5376	2072	0.5376
2	1	1253	0.3251	3325	0.8627
3	2	338	0.0877	3663	0.9504
4	3	115	0.0298	3778	0.9803
5	4	46	0.0119	3824	0.9922
6	5	19	0.0049	3843	0.9971
7	6	6	0.0016	3849	0.9987
8	7	2	0.0005	3851	0.9992
9	10	3	0.0008	3854	1

El **86%** de las compilaciones tuvieron a lo mucho un error, y el **54%** no tuvo ningún error.

En la siguiente figura, se muestra una comparación de compilaciones exitosas con las que no funcionaron.

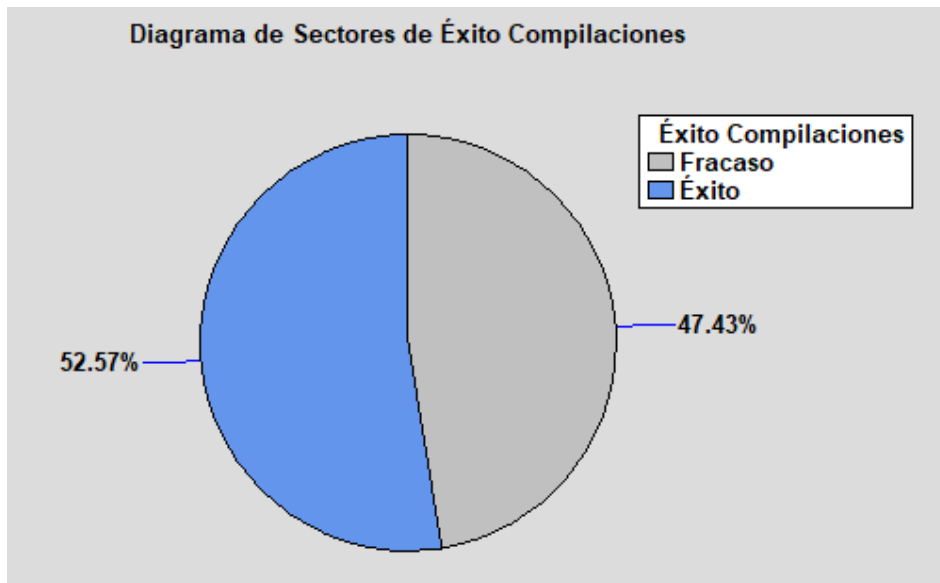


Fig. 33. Comparación porcentual de compilaciones exitosas con las falladas.

Se ha registrado que casi el 53% de las compilaciones fueron exitosas. En realidad, este valor debería ser más alto debido a un bug que causaba, durante un par de días, que cada compilación se registre como fallada.

Se cree que hay un posible vínculo entre la cantidad de compilaciones realizadas por usuario y el porcentaje de errores obtenidos. La hipótesis es que en general, mientras más compilaciones, menor será el porcentaje de errores debido al aprendizaje del código y las minuciosidades del lenguaje. Haciendo un análisis relacional de regresión simple (lineal) o correlación entre estos datos, se puede obtener el siguiente modelo:

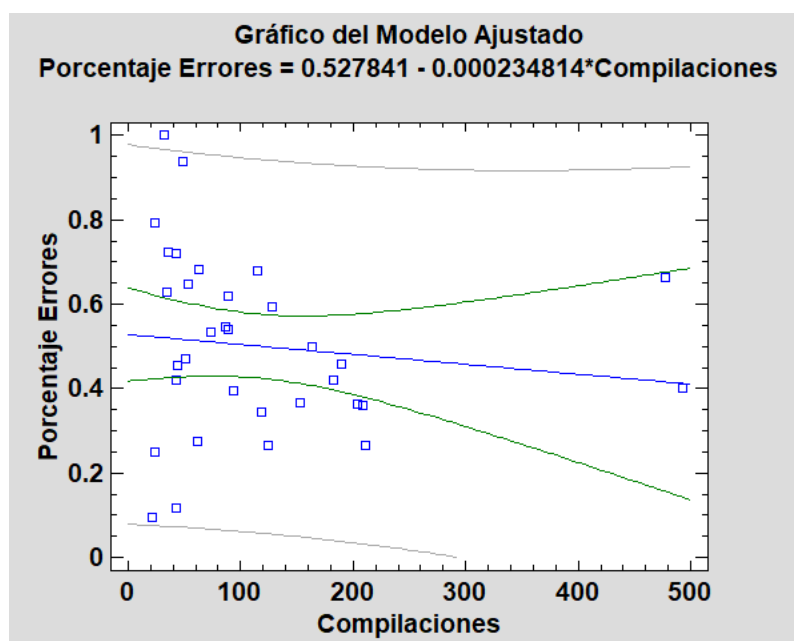


Fig. 34. Gráfica del modelo de regresión lineal entre porcentaje de errores vs compilaciones de los usuarios.

Aunque no se puede calcular con exactitud, se puede aplicar la siguiente ecuación para tratar de predecir el porcentaje de errores, basado en las compilaciones: $\text{Porcentaje Errores} = 0.527841 - 0.000234814 * \text{Compilaciones Por Usuario}$

El valor de la correlación lineal de Pearson es -0.124079. Pese a la posibilidad de una posible correlación, se muestra que no hay una relación estadísticamente significativa entre “Porcentaje Errores” y “Compilaciones por Usuario”, lo cual descarta la relación.

Al analizar los tipos de errores detectados, se puede notar que casi el 90% de los errores son sintácticos.

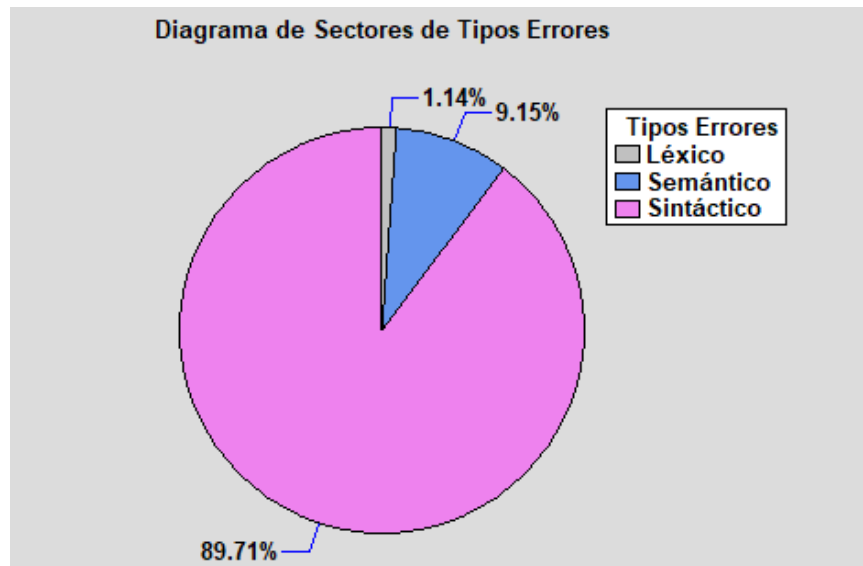


Fig. 35. Comparación porcentual de tipos de errores detectados.

TABLA 3.28 Tabla de frecuencias de los tipos de errores detectados.

Tabla de Frecuencia para Tipos Errores						
Clase	Valor	Frecuencia	Frecuencia Relativa	Frecuencia Acumulada	Frecuencia Rel. acum.	
1	Léxico	30	0.0114	30	0.0114	
2	Semántico	241	0.0915	271	0.1029	
3	Sintáctico	2362	0.8971	2633	1	

3.5 – Conclusión de Resultados

Según los datos recolectados de las encuestas de diferentes usuarios y participantes, se apunta con gran certeza a la posibilidad de implementar el Compilador Cándor como una herramienta para el aprendizaje inicial de programación a nivel de bachillerato o incluso universitario. La mayoría de los usuarios, mayormente estudiantes de informática a nivel de bachillerato y universidad concuerdan en este aspecto.

Validando el comportamiento temporal de la eficiencia de desempeño, los usuarios dieron un puntaje de **76.25%** en la subcaracterística de rendimiento. Para un inicio, este nivel de rendimiento es suficiente para muchos usuarios que tengan equipos con características promedias. Este aspecto se podría mejorar mediante una mejor optimización de código, paralelismo y uso multihilo, reducción de demoras arbitrarias y carga única (no repetida) de archivos especiales requeridos. La diferencia no sería tan notable en equipos más potentes, pero sí en máquinas más antiguas. En la utilización de recursos, se comprobó que los recursos requeridos no exceden límites aceptables para cualquier máquina suficientemente poderosa para correr Windows 8.1 o Windows 10 (los sistemas operativos compatibles con el compilador).

Validando la adecuación funcional, la completitud funcional tiene un porcentaje de **79.8%**, lo cual es más que suficiente como un inicio, pero se podría ir agregando funcionalidades adicionales complementarias en el futuro. La corrección funcional se encuentra con **88.25%**; sería difícil aumentar notablemente este valor. Por último, la pertinencia funcional tiene un puntaje de **84.82%**. Es posible incrementar este valor a través de mejoras a la tabla de errores, manuales y ejercicios acompañantes.

Los usuarios que más usaron el compilador tuvieron un promedio de **50%** de éxito en sus compilaciones. Considerando que ninguno de los usuarios recibió entrenamiento formal sobre el uso y manipulación del compilador, esta es una buena cifra. Se podría mejorar este dato mediante clases de entrenamiento para nuevos usuarios y la incorporación de mejor documentación acompañante.

Entre la completitud del Compilador Cóndor y el resumen presentado de datos, se puede comprobar un cumplimiento fiel y exitoso de los objetivos propuestos al inicio del proyecto.

CONCLUSIONES

- Mantener las palabras reservadas independientes de un autómata finito determinista (AFD) permitió mantenerlo relativamente pequeño, fácil de manejar y sin necesidad de técnicas extensivas de optimización.
- Entre los analizadores sintácticos existentes se escogió usar un ascendente LALR(1) por su amplitud de lenguajes aceptados, buen rendimiento y detección rápida de errores. La dificultad de construcción del autómata de pila (AP) se anula con herramientas contemporáneas y capacidades computacionales modernas.
- En base a la evolución notable de los lenguajes de programación hacia los lenguajes naturales en las décadas pasadas, no existe ningún impedimento para que se pueda crear y usar lenguajes en diferentes idiomas, al menos en un contexto pedagógico.
- La construcción de un nuevo lenguaje de programación, independientemente del idioma de las palabras reservadas, es una tarea muy difícil que requiere tiempo, planificación, periodo de pruebas y entendimiento extensivos para llevar a cabo. A pesar de las muchas horas pasadas en repasar la teoría, realizar ejercicios e ingeniar un nuevo lenguaje, hubo algunos inconvenientes durante el desarrollo y efectos inesperados tras la culminación del proyecto.
- La implementación de pequeñas funcionalidades y características que faciliten el proceso de desarrollo de código y extirpación de errores puede llegar lejos; los usuarios notan estas pequeñas mejoras y los aprecian.
- El uso de un sistema de actualizaciones automáticas es la manera óptima de distribuir nuevas versiones de un programa a los usuarios. Este método implica que no es necesario contactar a cada usuario y pedir que todos descarguen la nueva versión. Además, la actualización es rápida, no detiene la ejecución del programa y no requiere la interacción de parte del usuario.
- Las personas que usaron y vieron el funcionamiento del compilador acordaron en gran parte la facilidad de uso y creación de código. Además, estuvieron de acuerdo en la posibilidad de usarlo como una herramienta educativa como un primer lenguaje de programación. Basándose en las respuestas de los usuarios y la ratio de buenas compilaciones de código, el proceso de aprendizaje del **Compilador Cóndor** no es extenso ni requiere un gran esfuerzo para llegar a desarrollar programas funcionales.
- Refiriéndose al comportamiento de tiempo dentro de la eficiencia de desempeño, el Compilador Cóndor no tiene un rendimiento excelente, pero sí dentro de un rango aceptable.

- Los participantes contestaron mayormente de forma positiva acerca del uso de características multimediales en la programación. Basándose en este dato y en el estudio realizado por Rubio, Romero-Zaliz, Mañoso y De Madrid en el año 2015, lleva a la conclusión de que la enseñanza de la materia de programación no se debe centrar únicamente en teoría y ejercicios comunes de lógica; el uso de sonidos, gráficos y planteamientos creativos puede aumentar el interés y el aprendizaje en algunos estudiantes y ayudar a cerrar la brecha de género existente en carreras que involucran la programación.

RECOMENDACIONES

- Para una mejor validación de los méritos de emplear el Compilador Cóndor en enseñar programación en los colegios, sería recomendable hacer un estudio comparativo entre dos paralelos, enseñando el uso y lenguaje del Compilador Cóndor en uno y otro lenguaje convencional, como Python, C# o Java, en el otro.
- El compilador requiere pocos cambios para funcionar en cualquier idioma. Se sugiere subir el proyecto a un repositorio público, como GitHub, permitiendo que usuarios envíen mejoras y traducciones para otros idiomas.
- Desarrollar un lenguaje no es una tarea sencilla; se sugiere que haya un equipo de varias personas para crear, probar, usar y aprender bien el lenguaje que se esté desarrollando. Además, conviene buscar maneras de que el lenguaje sea aún más inteligible para nuevos aprendices.
- Se podría considerar crear un proyecto similar en un lenguaje como Java para que funcione en diferentes sistemas operativos. Esto ayudaría a fomentar un ambiente de educación libre y gratuita, ya que los usuarios podrían usar sistemas operativos abiertos como Linux.

REFERENCIAS

- Aho, A. V. (Columbia U., Lam, M. S. (Stanford U., Sethi, R. (Avaya), & Ullman, J. D. (Stanford U. (2007). *Compilers Principles, Techniques, & Tools*. In M. Hirsch, M. Goldstein, H. Katherine, & J. Holcomb (Eds.), *Vasa* (2nd ed.). Pearson Addison Wesley.
- Ali, A., & Smith, D. (2014). Teaching an introductory programming language in a general education course. In *Journal of Information Technology Education: Innovations in Practice* (Vol. 13). Retrieved from <http://www.jite.org/documents/Vol13/JITEv13IIPp057-067Ali0496.pdf>
- Brookshear, J. G. (1993). *Teoría de la Computación: Lenguajes formales, autómatas y complejidad*. Wilmington: The Benjamin/Cummings Publishing Company, Inc.
- Ceruzzi, P., & Wexelblat, R. L. (1984). History of Programming Languages. In R. L. Wexelblat (Ed.), *Technology and Culture* (Vol. 25). Academic Press.
<https://doi.org/10.2307/3104692>
- Chen, C., Haduong, P., Brennan, K., Sonnert, G., & Sadler, P. (2019). The effects of first programming language on college students' computing attitude and achievement: a comparison of graphical and textual languages. *Computer Science Education*, 29(1), 23–48. <https://doi.org/10.1080/08993408.2018.1547564>
- Chouchanian, V. (2010). Programming languages. Retrieved July 12, 2021, from <http://www.csun.edu/~vgc30838/Projecth.html>
- Crow, T., Kirk, D., Luxton-Reilly, A., & Tempero, E. (2020). Teacher perceptions of feedback in high school programming education: A thematic analysis. *ACM International Conference Proceeding Series*. Association for Computing Machinery.
<https://doi.org/10.1145/3421590.3421595>
- Ferguson, A. (2004). A History of Computer Programming Languages. Retrieved July 12, 2021, from http://cs.brown.edu/~adf/programming_languages.html
- Garrido Sánchez, J. F. (2011). *Compiladores*. Ibarra: Universidad Técnica del Norte.
- Harris, J. (Georgia S. U. (2011). TEACHING GAME PROGRAMMING USING XNA: WHAT WORKS AND WHAT DOESN'T. *Article in Journal of Computing Sciences in Colleges*, 27(2), 174–181. Retrieved from <https://www.researchgate.net/publication/260146742>
- Hawi, N. (2010). Causal attributions of success and failure made by undergraduate students in an introductory-level computer programming course. *Computers and Education*,

54(4), 1127–1136. <https://doi.org/10.1016/j.compedu.2009.10.020>

- International Organization for Standardization. (2017). The ISO/IEC 25000 series of standards. *ISO/IEC 25010:2011 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models*, 1, 34. Retrieved from <https://iso25000.com/index.php/en/iso-25000-standards%0Ahttps://iso25000.com/index.php/normas-iso-25000/iso-25010/20-adecuacion-funcional>
- Kaleliolu, F. (2015). A new way of teaching programming skills to K-12 students: Code.org. *Computers in Human Behavior*, 52, 200–210. <https://doi.org/10.1016/j.chb.2015.05.047>
- Kirk, D., Tempero, E., Luxton-Reilly, A., & Crow, T. (2020). High School Teachers' Understanding of Code Style. *ACM International Conference Proceeding Series*. Association for Computing Machinery. <https://doi.org/10.1145/3428029.3428047>
- Limited, I. E. S. (2012). *Principles of Compiler Design (Express Learning)*. New Delhi: Dorling Kindersley Pvt. Ltd.
- Maurya, R. K. (2011). *Compiler Design* (2011th ed.). New Delhi: Dreamtech Press.
- MIRA Limited. (2004). *MISRA-C:2004 Guidelines for the use of the C language in critical* (2nd ed.). Nuneaton: MIRA Limited.
- Mladenović, M., Mladenović, S., & Žanko, Ž. (2020). Impact of used programming language for K-12 students' understanding of the loop concept. *International Journal of Technology Enhanced Learning*, 12(1), 79–98. <https://doi.org/10.1504/IJTEL.2020.103817>
- Motor Industry Software Reliability Association. (2016). *MISRA C : 2012 Amendment 1 Additional security guidelines for MISRA C : 2012*. Nuneaton: HORIBA MIRA Limited.
- Prokofyeva, N., Uhanova, M., Katalnikova, S., Synytsya, K., & Jurenoks, A. (2016). Introductory Programming Training of First Year Students. *Procedia Computer Science*, 104, 286–293. Elsevier B.V. <https://doi.org/10.1016/j.procs.2017.01.137>
- Rubio, M. A., Romero-Zaliz, R., Mañoso, C., & De Madrid, A. P. (2015). Closing the gender gap in an introductory programming course. *Computers and Education*, 82, 409–420. <https://doi.org/10.1016/j.compedu.2014.12.003>
- Sanchis Llorca, F. J. (IBM), & Galán Pascual, C. (C. T. N. E. . (1986). *Compiladores Teoría y Construcción*. Madrid: Paraninfo.
- Steinberg, J. (2012). *"Hello, World!" A History of Programming*. CreateSpace Independent Publishing Platform.

- Tsukamoto, H., Takemura, Y., Nagumo, H., Ikeda, I., Monden, A., & Matsumoto, K. I. (2015). Programming education for primary school children using a textual programming language. *Proceedings - Frontiers in Education Conference, FIE, 2015*. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/FIE.2015.7344187>
- Tuparov, G., Tuparova, D., & Tsarnakova, A. (2012). Using Interactive Simulation-Based Learning Objects in Introductory Course of Programming. *Procedia - Social and Behavioral Sciences, 46*, 2276–2280. <https://doi.org/10.1016/j.sbspro.2012.05.469>
- Weintrop, D., & Wilensky, U. (2019). Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers and Education, 142*, 103646. <https://doi.org/10.1016/j.compedu.2019.103646>
- Yadin, A. (2011). Reducing the dropout rate in an introductory programming course. *ACM Inroads, 2*(4), 71–76. <https://doi.org/10.1145/2038876.2038894>
- Zúñiga, A., Sierra, G., Bel-Enguix, G., & Galicia-Haro, S. N. (2018). Towards a natural language compiler. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 11289 LNAI*, 70–82. Springer Verlag. https://doi.org/10.1007/978-3-030-04497-8_6

ANEXOS

Anexo A: Lexemas reconocidos por el analizador léxico

TABLA 4.1 Lista de lexemas reconocidos en el lenguaje.

#	Símbolo	Nombre	Patrón	Descripción
1	i	Identificador	letra(letra digito _)*	Nombre de variable, o palabra reservada
2	+	operadorsuma	+	Operador aritmético que suma dos operandos
3	-	operadorresta	-	Operador aritmético que resta dos operandos
4	*	operadormultiplicacion	*	Operador aritmético que multiplica dos operandos
5	/	operadordivision	/	Operador aritmético que divide dos operandos
6	^	operadorpotencia	^	Operador aritmético que eleva un operando a la potencia de otro
7	&	operadorand	&	Operador lógico que evalúa la veracidad de dos expresiones
8		operadoror		Operador lógico que evalúa la veracidad de al menos una de dos expresiones
9	<	menorque	<	Operador relacional que evalúa si la primera expresión es menor que la segunda
10	≤	menorigualque	<=	Operador relacional que evalúa si la primera expresión es menor o igual que la segunda
11	=	asignacion	=	Operador aritmético que asigna el valor del segundo operando al primero
12	¿	igualque	==	Operador relacional que evalúa si la primera expresión es igual que la segunda
13	>	mayorque	>	Operador relacional que evalúa si la primera expresión es mayor que la segunda
14	≥	mayorigualque	>=	Operador relacional que evalúa si la primera expresión es mayor o igual que la segunda
15	#	diferenteque	!=	Operador relacional que evalúa si la primera expresión es diferente que la segunda
16	%	negacion	!	Operador lógico que invierte el resultado de una expresión
17	a	incremento	++	Operador aritmético (unario) que incrementa por uno un operando
18	d	decremento	--	Operador aritmético (unario) que decrementa por uno un operando
19	[corcheteizq	[Usado al lado derecho de un arreglo, seguido por el índice (o los índices) deseado
20]	corcheteder]	Usado al lado derecho de un arreglo, precedido por el índice (o los índices) deseado
21	p	puntoycoma	;	Usado en bucles para, delimitando sus partes

22	c	coma	,	Usado para separar varios parámetros o argumentos
23	~	dospuntos	:	Usado para indicar el inicio de funciones, estructuras de control y bucles, tras su declaración
24	(parentesisizq	(Usado al lado derecho de llamadas a métodos, para especificar el orden de ejecución de operaciones, para retornar un valor de una función, y para la declaración de funciones, estructuras de control y bucles, seguido por los posibles argumentos, parámetros o expresiones requeridos
25)	parentesisder)	Usado al lado derecho de llamadas a métodos, para especificar el orden de ejecución de operaciones, para retornar un valor de una función, y para la declaración de funciones, estructuras de control y bucles, precedido por los posibles argumentos, parámetros o expresiones requeridos
26	l	literaldecimal	(+ -)?(1-9)digito*.digito+ (+ -)?0.digito+	Un número con precisión decimal
27	l	literalentero	(+ -)?(1-9)digito* 0	Un número sin precisión decimal
28	l	literalcadena	"(letra digito simbolos)*"	Un conjunto de caracteres, formando una cadena de texto
29	l	literalchar	'letra digito simbolos'	Un carácter alfanumérico o simbólico

Anexo B: Palabras reservadas

TABLA 4.2 Lista de palabras reservadas.

Palabra	Descripción
entero	Tipo de dato entero. (int en C#)
ent	
decimal	Tipo de dato decimal. (double en C#)
dec	
carácter	Tipo de dato carácter. (char en C#)
car	
texto	Tipo de dato de cadena de text. (string en C#)
tex	
lógico	Tipo de dato lógico. (bool en C#)
log	
verdad	Literal de tipo booleano
falso	Literal de tipo booleano
textura	Tipo de dato que representa un archivo gráfico.
ras	
col	Tipo de dato que representa un color (Color en C#)
color	
si	Estructura de control If
sinosi	Estructura de control else if
sino	Estructura de control else
selección	Estructura de control switch
caso	Parte de la estructura de control switch, case
predeterminado	El caso seleccionado por defecto (default)
fin	Fin de cualquier estructura de control, bucle, o método
mientras	Bucle while
hacer	Bucle do
repetir	Bucle que repite las veces requeridas
para	Bucle for
salir	Sale de un bucle (break)
nada	Tipo de dato retornado (void)
retornar	Retorna un dato de un método
dibujar	Parte del programa que permite dibujar objetos en la pantalla
negro	
azul	
verde	
celeste	
rojo	Color
rosado	
amarillo	
blanco	
pi	Valor de pi

intentar	Bloque de código que se debe intentar
error	Bloque de código que se ejecuta en caso de un error en un bloque previo intentar

Anexo C: Funciones preprogramadas

TABLA 4.3 Lista de funciones preprogramadas.

Palabra	Descripción	Entradas	Salida	Resultado
sen	Función que calcula seno	A: Decimal - radianes	Función de seno sobre A (Decimal)	
cos	Función que calcula coseno	A: Decimal - radianes	Función de coseno sobre A (Decimal)	
tan	Función que calcula tangente	A: Decimal - radianes	Función de tangente sobre A (Decimal)	
potencia	Función que calcula potencia	A: Número decimal a elevar, B: número decimal de exponente	A^B (Decimal)	
esperar	Función que espera el tiempo deseado	A: Entero - milisegundos	Ninguna	La ejecución del programa se detiene el tiempo ingresado
raízcuadrada	Función que calcula la raíz cuadrada	A: Decimal	Función de raíz cuadrada sobre A (Decimal)	
leer	Función que lee de la consola	Ninguna	Texto dentro de la línea de la consola (texto)	
escribir	Función que escribe a la consola	A: Texto	Ninguna	El texto indicado se escribe en la consola
convertir	Función que convierte de un tipo de dato a otro	A: (variable o literal de cualquier tipo de dato)	Devuelve la conversión del tipo de dato al tipo detectado automáticamente	
reproducir	Función que reproduce un archivo de audio .wav	A: Texto - Ubicación del archivo de audio a reproducir	Ninguna	Se reproduce el archivo de audio
sonido	Función que hace un sonido	A: Entero - frecuencia del sonido, B: Entero - duración del sonido en milisegundos	Ninguna	Se produce un sonido a frecuencia A durante duración B

leerras	Función que importa un archivo gráfico	A: Text - Ubicación del archivo gráfico a importar	Textura	
leertexto	Función que lee todo el texto de un archivo	A: Texto - Ubicación del archivo de texto a leer	B: Texto - Todo el texto del archivo	
escribirtexto	Función que escribe texto a un archivo	A: Texto - El texto a escribir, B: Texto - La ubicación del archivo de texto	Ninguna	Escribe el texto indicado a un archivo de texto
dibujarras	Función que dibuja una textura en la pantalla	A: Archivo gráfico , B: Decimal - Coordenada X, C: Decimal - Coordenada Y, D: Color	Ninguna	Dibuja una textura 2D en la pantalla en las coordenadas dadas
longitud	Función que calcula la longitud de un objeto (texto, arreglo)	A: Texto o Arreglo	A: Entero - La cantidad de elementos que contenga la entrada	
azar	Función que devuelve un valor aleatorio	A: Ninguna	A: Entero - Un número entero positivo cualquier	
		A: Entero - máximo	A: Entero - Un número entero entre cero (inclusivo) y el número máximo (exclusivo)	
		A: Entero - mínimo, B: Entero - máximo	A: Entero - Un número entero entre el número mínimo (inclusivo) y el número máximo (exclusivo)	
convertirEnt	Función que devuelve un valor entero		A: Entero	
convertirDec	Función que devuelve un valor decimal		A: Decimal	
convertirTex	Función que devuelve un valor texto	A: Dato a convertir	A: Texto	
convertirCar	Función que devuelve un valor carácter		A: Carácter	
convertirLog	Función que devuelve un valor lógico		A: Lógico	

Anexo D: Reglas gramaticales reconocidas por el analizador sintáctico

TABLA 4.4 Lista de reglas gramaticales.

#	Lado Izquierdo	Lado Derecho
1	<programa>	<funciones><sentencias><dibujar>
2	<programa>	<funciones><sentencias>
3	<programa>	<sentencias><dibujar>
4	<programa>	<sentencias>
5	<funciones>	<funcion><funciones>
6	<funciones>	<funcion>
7	<función>	funcion <tipo_dato> identificador (<parametros>) : <sentencias>
8	<dibujar>	dibujar : <sentencias>
9	<tipo_dato>	entero
10	<tipo_dato>	decimal
11	<tipo_dato>	texto
12	<tipo_dato>	carácter
13	<tipo_dato>	lógico
14	<tipo_dato>	textura ráster
15	<tipo_dato>	color
16	<tipo_dato>	nada
17	<parametros>	<parametro>,<parametros>
18	<parametros>	<parametro>
19	<parametro>	<tipo_dato>identificador
20	<sentencias>	<instrucción><sentencias>
21	<sentencias>	<instrucción> fin
22	<instruccion>	<declaracion>
23	<instruccion>	<asignacion>
24	<instruccion>	<cambio_unitario>
25	<instruccion>	<si>
26	<instruccion>	<seleccion>
27	<instruccion>	<mientras>
28	<instruccion>	<hacer>
29	<instruccion>	<repetir>
30	<instruccion>	<para>
31	<instruccion>	<retornar>
32	<instruccion>	<intentar>
33	<instruccion>	salir
34	<instruccion>	<llamado_metodo>
35	<declaracion>	<tipo_dato><lista_declaraciones>
36	<declaracion>	<tipo_dato> identificador[<argumentos>]
37	<lista_declaraciones>	<una_asignacion>,<lista_declaraciones>
38	<lista_declaraciones>	<una_asignacion>
39	<una_asignacion>	identificador
40	<una_asignacion>	<asignacion>
41	<asignacion>	identificador=<condiciones>

42	<asignacion>	identificador=<expresion>
43	<cambio_unitario>	identificador++
44	<cambio_unitario>	identificador--
45	<condiciones>	<condicion><operador_logico><condiciones>
46	<condiciones>	(<condicion><operador_logico><condiciones>)
47	<condiciones>	<condicion>
48	<condicion>	<cond>
49	<condicion>	! <cond>
50	<cond>	<expresion><relacion><expresion>
51	<cond>	verdad
52	<cond>	falso
53	<expresion>	<expresion>+<termino>
54	<expresion>	<expresion>-<termino>
55	<expresion>	<termino>
56	<term>	<termino>*<factor>
57	<term>	<termino>/<factor>
58	<term>	<termino>^<factor>
59	<term>	<factor>
60	<factor>	identificador
61	<factor>	identificador[<argumentos>]
62	<factor>	(<expresion>)
63	<factor>	literal
64	<factor>	<llamado_metodo>
65	<relacion>	==
66	<relacion>	!=
67	<relacion>	<=
68	<relacion>	<
69	<relacion>	>
70	<relacion>	>=
71	<operador_logico>	&
72	<operador_logico>	
73	<si>	si(<condiciones>) : <sentencias><lista_sinosi>
74	<si>	si(<condiciones>) : <sentencias>
75	<lista_sinosi>	sinosi(<condiciones>): <sentencias><lista_sinosi>
76	<lista_sinosi>	sinosi(<condiciones>): <sentencias>
77	<lista_sinosi>	sino: <sentencias>
78	<seleccion>	seleccion(identificador) : <casos>
79	<casos>	<caso><casos>
80	<casos>	<caso><predeterminado>
81	<casos>	<caso>
82	<caso>	caso literal: <sentencias>
83	<predeterminado>	predeterminado: <sentencias>
84	<mientras>	mientras(<condiciones>) : <sentencias>
85	<hacer>	hacer: <sentencias> mientras(<condiciones>)
86	<repetir>	repetir : <sentencias> (<expresion>)

87	<para>	para(<inipara><paracondiciones><paraasignacion> :
		<sentencias>
88	<inipara>	<tipo_dato><asignacion>;
89	<inipara>	identificador;
90	<inipara>	;
91	<paracondiciones>	<condiciones>;
92	<paracondiciones>	;
93	<paraasignacion>	<asignacion>)
94	<paraasignacion>	<cambio_unitario>)
95	<paraasignacion>	<expresion>)
96	<paraasignacion>)
97	<retornar>	retornar (<expresion>)
98	<retornar>	retornar ()
99	<probar>	probar : <sentencias> error: <sentencias>
100	<llamado_metodo>	identificador(<argumentos>)
101	<argumentos>	<expresion>,<argumentos>
102	<argumentos>	<expresion>
103	<funcion>	funcion <tipo_dato> identificador () : <sentencias>
104	<llamado_metodo>	identificador()
105	<instruccion>	<asignacion_array>
106	<asignacion_array>	identificador[<argumentos>]=<condiciones>
107	<asignacion_array>	identificador[<argumentos>]=<expresion>

Anexo E: Reglas y validaciones del analizador semántico

TABLA 4.5 Lista de reglas y validaciones semánticas.

#	Regla	Código	Tipo	Verificación de Tipo	Adicional
1	P' -> P	P'.Code = P.Code			
2	P -> F S D	P.Code = F.Code + S.Code + D.Code			
3	P -> F S	P.Code = F.Code + S.Code + D.Code			
4	P -> S D	P.Code = S.Code + D.Code			
5	P -> S	P.Code = S.Code			
6	F -> M F	F.Code = M.Code + F.Code			
7	F -> M	F.Code = M.Code			
8	M -> { T i (B) ~ } S	M.Code = T.Code + i.Code + "(" + B.Code + ")\n{" + S.Code			AddTDSMethod(T.Type, i, B)
9	D -> ö ~ } S	D.Code = "protected override void Draw(GameTime gameTime)\n{" + S.Code			
10	T -> e	T.Code = e.Code		T.Type = e.Type	
11	T -> b	T.Code = b.Code		T.Type = b.Type	
12	T -> t	T.Code = t.Code		T.Type = t.Type	
13	T -> g	T.Code = g.Code		T.Type = g.Type	
14	T -> h	T.Code = h.Code		T.Type = h.Type	
15	T -> r	T.Code = r.Code		T.Type = r.Type	
16	T -> k	T.Code = k.Code		T.Type = k.Type	
17	T -> ë	T.Code = ë.Code		T.Type = ë.Type	
18	B1 -> É c B2	B1.Code = É.Code + "," + B2.Code			
19	B -> É	B.Code = É.Code			
20	É -> T i	É.Code = T.Code + i.Code			
21	S -> I } S	S.Code = I.Code + "\n" + S.Code			
22	S -> I } f	S.Code = I.Code + "\n}"			
23	I -> G	I.Code = G.Code + ";"			

24	I	-> A	I.Code = A.Code + ";"		
25	I	-> C	I.Code = C.Code + ";"		
26	I	-> Í	I.Code = Í.Code		
27	I	-> Ó	I.Code = Ó.Code		
28	I	-> W	I.Code = W.Code		
29	I	-> H	I.Code = H.Code + ";"		
30	I	-> Y	I.Code = Y.Code		
31	I	-> Z	I.Code = Z.Code		
32	I	-> \	I.Code = \.Code + ";"		
33	I	-> _	I.Code = _.Code		
34	I	-> s	I.Code = "break;"		
35	I	-> `	I.Code = ` .Code + ";"		
36	G	-> T L	G.Code = T.Code + L.Code		InsertTDS(T.Type, L)
			G.Code = T.Code + "[" +		
			PrintMatrixDimensions(Á) + "]" + i.Code + "=		
37	G	-> T i [Á]	new " + T.Type + "[" + Á.Code + "]"	CheckTypes(int, Á)	InsertTDSArray(T.Type, i, Á)
38	L	-> U c L	L.Code = U.Code + "," + L.Code		
39	L	-> U	L.Code = U.Code		
40	U	-> i	U.Code = i.Code		
41	U	-> A	U.Code = A.Code		
42	A	-> i = E	A.Code = i.Code + "=" + E.Code	CheckType(GetTDSType(i), E)	
43	A	-> i = K	A.Code = i.Code + "=" + K.Code	CheckType(GetTDSType(i), bool)	
44	C	-> i a	C.Code = i.Code + "++"	CheckType(GetTDSType(i), decimal)	
45	C	-> i d	C.Code = i.Code + "--"	CheckType(GetTDSType(i), decimal)	
46	K	-> Q O K	K.Code = Q.Code + Ok.Code + K.Code	K.Type = bool	

47	K	-> (Q O K)	K.Code = "(" + Q.Code + O.Code + K.Code + ")"	K.Type = bool		
48	K	-> Q	K.Code = Q.Code	K.Type = bool		
49	Q	-> X	Q.Code = X.Code			
50	Q	-> % X	Q.Code = "!" + X.Code			
51	X	-> E R E	X.Code = E.Code + R.Code + E.Code		CheckType(E.Type, E.Type)	
52	X	-> v	X.Code = v.Code	X.Type = bool		
53	X	-> q	X.Code = q.Code	X.Type = bool		
54	E1	-> E2 + J	E1.Code = E2.Code + J.Code	E1.Type = TypeMerger(E2, J)	CheckType(E2, J)	
55	E1	-> E2 - J	E1.Code = E2.Code - J.Code	E1.Type = TypeMerger(E2, J)	CheckType(E2, J)	
56	E	-> J	E.Code = J.Code	E.Type = J.Type		
57	J1	-> J2 * N	J1.Code = J2.Code + "*" + N.Code	J1.Type = TypeMerger(J2.Type, N.Type)	CheckType(decimal, J2) CheckType(decimal, N)	
58	J1	-> J2 / N	J1.Code = J2.Code + "/" + N.Code	J1.Type = TypeMerger(J2.Type, N.Type)	CheckType(decimal, J2) CheckType(decimal, N)	
59	J1	-> J2 ^ N	J1.Code = "Math.Pow(" + J2.Code + "," + N.Code + ")"	J1.Type = decimal	CheckType(decimal, J2) CheckType(decimal, N)	
60	J	-> N	J.Code = N.Code	J.Type = N.Type		
61	N	-> i	N.Code = i.Code	N.Type = GetTDSType(i)		
62	N	-> i [Á]	N.Code = i.Code + "[" + Á.Code + "]"	N.Type = GetTDSType(i)	CheckTypes(int, Á)	CheckDimensions(I, Á)
63	N	-> (E)	N.Code = "(" + E.Code + ")"	N.Type = E.Type		
64	N	-> l	N.Code = l.Code	N.Type = l.Type		
65	N	-> `	N.Code = ` .Code	N.Type = ` .Type		

66	R	-> ¿	R.Code = "=="	
67	R	-> #	R.Code = "!="	
68	R	-> m	R.Code = ">="	
69	R	-> >	R.Code = ">"	
70	R	-> i	R.Code = "<="	
71	R	-> <	R.Code = "<"	
72	O	-> &	O.Code = "&&"	
73	O	-> o	O.Code = " "	
74	Í	-> u(K) ~ }SÑ	Í.Code = "if(" + K.Code + ")\n{" + S.Code + Ñ.Code	
75	Í	-> u(K) ~ }S	Í.Code = "if(" + K.Code + ")\n{" + S.Code	
76	Ñ	-> w(K) ~ }SÑ	Ñ1.Code = "else if(" + K.Code + ")\n{" + S.Code + Ñ2.Code	
77	Ñ	-> w(K) ~ }S	Ñ.Code = "else if(" + K.Code + ")\n{" + S.Code	
78	Ñ	-> x ~ }S	Ñ.Code = "else\n{" + S.Code	
79	Ó	-> y(i) ~ } Úf	Ó.Code = "switch(" + i.Code + ")\n{" + Ú.Code + "}"	CheckTypes(i.Type, Ú)
80	Ú	-> VÚ	Ú.Code = V.Code + Ú.Code	
81	Ú	-> VX	Ú = V.Code + X.Code	
82	Ú	-> V	Ú = V.Code	
83	V	-> z l ~ }S	V.Code = "case " + l.Code + ":\n" + S.Code + "break;"	V.Type = l.Type
84	X	-> á ~ }S	X.Code = "default:\n" + S.Code	
85	W	-> é(K) ~ }S	W.Code = "while(" + K.Code + ")\n{" + S.Code	
86	H	-> í ~ }S é(K)	H.Code = "do\n{" + S.Code + " while(" + K.Code + "}"	
87	Y	-> ñ ~ }S(E)	Y.Code = "for(" + CreateTemp() + ";" + GetTemp(TempSize - 1) + "<" + E.Code + ")\n{" + S.Code	CheckType(int, E)

88	Z	-> ó (Ä Ë Ö ~ } S	Z.Code = "for(" + Ä.Code + Ë.Code + Ö.Code + "\n{" + S.Code		
89	Ä	-> T A p	Ä.Code = T.Code + A.Code + ";"		
90	Ä	-> i p	Ä.Code = i.Code + ";"		
91	Ä	-> p	Ä.Code = ";"		
92	Ë	-> K p	Ë.Code = K.Code + ";"		
93	Ë	-> p	Ë.Code = ";"		
94	Ö	-> A)	Ö.Code = A.Code + ")"		
95	Ö	-> C)	Ö.Code = C.Code + ")"		
96	Ö	-> E)	Ö.Code = E.Code + ")"		
97	Ö	->)	Ö.Code = ")"		
98	\	-> ° (E)	\.Code = "return" + E.Code	\.Type = E.Type	
99	\	-> ° ()	\.Code = "return"	\.Type = "void"	
100	_	-> ú ~ } S ü ~ } S	_.Code = "try\n{" + S1.Code + "catch\n{" + S2.Code		
101	`	-> i (Á)	`.Code = i.Code + "(" + Á.Code + ")"	`.Type = GetTDSType(i)	CheckMethodTypes(i, Á)
102	Á	-> E c Á	Á.Code = E.Code + "," + Á.Code		
103	Á	-> E	Á.Code = E.Code		
104	M	-> { T i () ~ } S			
105	`	-> i ()			
106	l	ï	l.Code = ï.Code + ";"		
107	ï	i [Á] = K	ï.Code = i.Code + "[" + Á.Code + "]" = " K.Code		CheckTypes(int, Á.Type) CheckTypes(i, K)
108	ï	i [Á] = E	ï.Code = i.Code + "[" + Á.Code + "]" = " E.Code		CheckTypes(int, Á.Type) CheckTypes(i, E)
104	i		i.Code = i.Attribute	i.Type = GetTDSType(i)	
105	e		e.Code = "int"	e.Type = int	
106	b		b.Code = "decimal"	b.Type = decimal	

107	t	t.Code = "string"	t.Type = string
108	g	g.Code = "char"	g.Type = char
109	h	h.Code = "bool"	h.Type = bool
110	r	r.Code = "Texture2D"	r.Type = Texture2D
111	k	k.Code = "Color"	k.Type = Color
112	ë	ë.Code = "void"	ë.Type = void
113	v	v.Code = "true"	v.Type = bool
114	q	q.Code = "false"	v.Type = bool
115	l	l.Code = l.Attribute	l.Type = Type

Anexo F: Operaciones permitidas en el lenguaje

TABLA 4.6 Operaciones permitidas.

Primer Operando	Operador Aritmético	Segundo Operando	Resultado
entero	+, -, *, /, ^, =	entero	entero
		carácter	
	+	decimal	decimal
decimal	+, -, *, /, ^, =	entero	
		decimal	decimal
	+	carácter	
texto	+	entero	
		decimal	
	+, =	carácter	texto
carácter	+, -, *, /, ^	entero	entero
		decimal	decimal
	+	texto	texto
	+, -, *, /, ^	carácter	entero
	=		carácter
lógico	=	lógico	lógico
textura	=	textura	textura
color	=	color	color

Anexo G: Lista de errores

TABLA 4.7 Lista de errores detectables por el compilador.

Código	Descripción
200	Ha escrito un símbolo inicial no válido (como punto, raya baja, etc.). Póngalo en el contexto correcto, o elimínelo.
205	Ha escrito una comilla simple seguida por otra. Debe insertar otro símbolo entre ambas comillas.
212	Empezó a escribir un número decimal, pero no lo completó. Elimine el punto o termínelo con un dígito.
213	Abrió una comilla simple. Debe existir exactamente un carácter y otra comilla simple después de la primera.
214	Empezó a escribir un número decimal (positivo o negativo), pero no lo completó. Elimine el punto o termínelo con un dígito.
301	Token inicial inválido. Puede empezar con: función, identificador, tipo de dato, si, selección, mientras, hacer, repetir, para, intentar.
302	Se espera el final del código. No debe haber más.
303	Token intermedio encontrado. Se espera el inicio de una nueva instrucción.
304	Se espera: la sección dibujar, el final del código. Tokens intermedios encontrados después de fin.
305	Se espera: una nueva línea. Otro token hallado.
306	Se espera: un tipo de dato. Se encontró otro token.
307	Se espera: el nombre de un identificador. Otro token hallado.
308	Se espera: (, [, =, ++, --. Se encontró otro token.
309	Se espera: (. Se encontró otro token.
310	Se espera: :. Otro token hallado.
311	Se espera: el inicio de una nueva sentencia, fin. Tokens intermedios hallados.
312	Se espera: nueva línea, coma, [, =. Otro token encontrado.
313	Se espera: nueva línea, coma. Se encontró otro token.
314	Se espera: identificador, (!, verdad, falso, literal, predeterminado. Otro token hallado.
315	Se espera:), fin de línea. Se encontró otro token.
316	Se espera: identificador, (,), literal. Otro token hallado.
317	Se espera: identificador, (, literal. Otro token hallado.
318	Se espera: identificador, tipo de dato, ;. Otro token hallado.
319	Se esperaba cualquier token excepto: :, [,], =, ++, --, !, operador aritmético, operador relacional.
320	Se espera:), fin de línea, coma, +, -, operador relacional, ;. Se encontró otro token.
321	Se espera:), fin de línea, coma, ;. Otro token hallado.
322	Se espera:), fin de línea, coma, fin,], cualquier tipo de operador, ;. Se encontró otro token.
323	Se espera:), fin de línea, coma, operador lógico, ;. Otro token hallado.
324	Se espera: identificador, (, verdad, falso, literal, predeterminado. Se encontró otro token.
325	Se espera: (,), fin de línea, coma, fin, [,], cualquier tipo de operador, ;. Otro token hallado.
326	Se espera:), fin de línea, coma, fin, operador lógico, ;. Se encontró otro token.
327	Se espera:). Otro token hallado.
328	Se espera:), coma,], +, -. Se encontró otro token.
329	Se espera:]. Se encontró otro token.
330	Se espera: +, -, operador relacional. Otro token hallado.

- 331 Se espera: identificador, (, !, verdad, falso, literal, predeterminado, ;. Se encontró otro token.
- 332 Se espera: ;. Otro token hallado.
- 333 Se espera:), +, -. Se encontró otro token.
- 334 Se espera:), tipo de dato. Otro token hallado.
- 335 Se espera: fin de línea, coma, =. Otro token hallado.
- 336 Se espera: un operador lógico. Se encontró otro token.
- 337 Se espera:), +, -, operador relacional. Otro token hallado.
- 338 Se espera: =. Otro token hallado.
- 339 Se espera: while. Se encontró otro token.
- 340 Se espera: bloque error. Se encontró otro token.
- 341 Se espera:), coma. Otro token hallado.
- 342 Se espera:), fin de línea, coma, fin, +, -, operador lógico, ;. Se encontró otro token.
- 343 Se espera:),]. Otro token hallado.
- 344 Se espera: (,), [, =, ++, --, operador aritmético. Se encontró otro token.
- 345 Se espera: fin de línea, +, -, operador relacional. Se encontró otro token.
- 346 Se espera: caso. Otro token hallado.
- 347 Se espera: fin de línea, sinosi, sino. Se encontró otro token.
- 348 Se espera: fin. Token intermedio hallado.
- 349 Se espera: identificador, (, fin, verdad, falso, literal, caso, predeterminado. Otro token hallado.
- 350 Se espera: literal. Se encontró otro token.
- 401 El token y operador usados no son compatibles con el tipo entero.
- 402 El token y operador usados no son compatibles con el tipo decimal.
- 403 El token y operador usados no son compatibles con el tipo texto.
- 404 El token y operador usados no son compatibles con el tipo carácter.
- 405 El token y operador usados no son compatibles con el tipo lógico.
- 406 El token y operador usados no son compatibles con el tipo textura.
- 407 El token y operador usados no son compatibles con el tipo color.
- 408 El token y operador usados no son compatibles con el tipo nulo.
- 409 Error inesperado de tipos de datos.
- 410 Las expresiones ingresadas para el arreglo deben ser de tipo entero.
- 411 Uso de variable antes de su declaración.
- 412 Las dimensiones del arreglo no cuadran.
- 413 La expresión ingresada para repetir deben ser de tipo entero.
- 414 Los argumentos ingresados no cuadran con los parámetros permitidos de la función.
- 415 Función no declarada.
- 416 Tipo de retorno incorrecto.
Expresión inválida. Para usar una expresión en una sentencia para, incluya el nombre de variable en la asignación inicial.
- 417 No funciona la depuración con modo de dibujo.
- 501 Error en la creación del arreglo.
- 502 División para cero.
- 503 Acceso a índice inexistente del arreglo.
- 504 Un valor incorrecto fue ingresado a la función.
- 505 Archivo inexistente (error E/S).
- 601 Error no encontrado.
-

Anexo H: Errores detectados durante el uso del compilador

El 100% de los errores léxicos detectados fueron de símbolo inicial inválido (200).

La clasificación de los errores sintácticos se muestra en la siguiente tabla. No hay un diagrama de sectores porque hay demasiados errores para representar en un solo diagrama.

TABLA 4.8 Tabla de los errores sintácticos detectados.

Tabla de Frecuencia para Errores Sintácticos						
Clase	Valor	Frecuencia	Frecuencia Relativa	Frecuencia Acumulada	Frecuencia Rel. acum.	
1	301	25	0.0106	25	0.0106	
2	302	49	0.0207	74	0.0313	
3	303	62	0.0262	136	0.0576	
4	304	51	0.0216	187	0.0792	
5	307	8	0.0034	195	0.0826	
6	308	239	0.1012	434	0.1837	
7	309	62	0.0262	496	0.21	
8	310	880	0.3726	1376	0.5826	
9	311	233	0.0986	1609	0.6812	
10	312	20	0.0085	1629	0.6897	
11	313	10	0.0042	1639	0.6939	
12	314	37	0.0157	1676	0.7096	
13	315	1	0.0004	1677	0.71	
14	316	3	0.0013	1680	0.7113	
15	317	174	0.0737	1854	0.7849	
16	319	4	0.0017	1858	0.7866	
17	320	1	0.0004	1859	0.787	
18	322	154	0.0652	2013	0.8522	
19	325	243	0.1029	2256	0.9551	
20	327	8	0.0034	2264	0.9585	
21	328	26	0.011	2290	0.9695	
22	330	15	0.0064	2305	0.9759	
23	331	12	0.0051	2317	0.9809	
24	332	12	0.0051	2329	0.986	
25	335	19	0.008	2348	0.9941	
26	336	5	0.0021	2353	0.9962	
27	338	3	0.0013	2356	0.9975	
28	339	4	0.0017	2360	0.9992	
29	344	2	0.0008	2362	1	

Los errores sintácticos se muestran clasificados en la figura 36 y la tabla 4.9 de acuerdo a su porcentaje de detección.

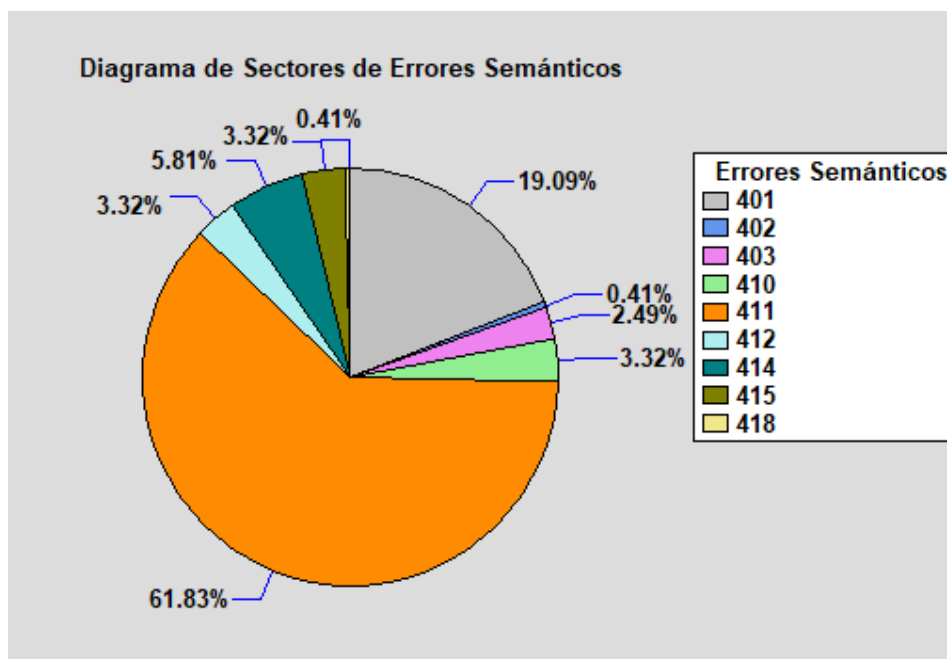


Fig. 36. Diagrama de los errores semánticos detectados.

TABLA 4.9 Tabla frecuencia de los errores semánticos detectados.

Tabla de Frecuencia para Errores Semánticos						
Clase	Valor	Frecuencia	Frecuencia Relativa	Frecuencia Acumulada	Frecuencia Rel. acum.	
1	401	46	0.1909	46	0.1909	
2	402	1	0.0041	47	0.195	
3	403	6	0.0249	53	0.2199	
4	410	8	0.0332	61	0.2531	
5	411	149	0.6183	210	0.8714	
6	412	8	0.0332	218	0.9046	
7	414	14	0.0581	232	0.9627	
8	415	8	0.0332	240	0.9959	
9	418	1	0.0041	241	1	

Anexo I: Código fuente del Compilador Cóndor

Código fuente del Compilador:

https://github.com/chessplayerjamesUTN/Condor_Compiler.git