UNIVERSIDAD NACIONAL MICAELA BASTIDAS DE APURÍMAC

FACULTAD DE INGENIERÍA

ESCUELA ACADÉMICO PROFESIONAL DE INGENIERÍA INFORMÁTICA Y SISTEMAS

DESARROLLO DE UN COMPILADOR PARA EL APRENDIZAJE DE MODULARIDAD MEDIANTE PSEUDOCÓDIGO EN ESPAÑOL – UNAMBA – 2015

TESIS

PRESENTADO POR:

MUÑOZ MIRANDA, Juan Carlos

PARA OPTAR EL TÍTULO PROFESIONAL DE INGENIERO INFORMÁTICO Y SISTEMAS

ABANCAY – APURÍMAC – PERÚ

2016

UNIVERSIDAD NACIONAL MICAELA BASTIDAS DE APURÍMAC
FACULTAD DE INGENIERÍAS

ESCUELA ACADÉMICO PROFESIONAL DE INGENIERÍA INFORMÁTICA Y SISTEMAS

TESIS

"DESARROLLO DE UN COMPILADOR PARA EL APRENDIZAJE DE MODULARIDAD MEDIANTE PSEUDOCÓDIGO EN ESPAÑOL – UNAMBA – 2015"

Presentada por **MUÑOZ MIRANDA JUAN CARLOS** a la Escuela Académica Profesional de Ingenierías, para optar el Título de:

INGENIERO INFORMÁTICO Y SISTEMAS

Sustentado y aprobado ante el jurado integrado por:

Presidente :

M.Sc. José Luis Merma Aroni

Primer Miembro :

Dra. Norma Lorena Catacora Flores

Segundo Miembro:

Ing. Marleny Peralta Ascue

Asesor:

M.Sc. Ing. Ecler Mamani Vilca

DEDICATORIA

A Dios que siempre está a mi lado, me guía, me enseña, me protege, me guarda, me cuida y me da fuerzas para seguir adelante.

A mi padre Sr. **Juan Muñoz Huachaca**, quien día a día se esfuerza para darme sustento, me aconseja sobre la vida de superación sobre cualquier circunstancia, así como su apoyo incondicional.

A mi madre Sra. **Beatriz Miranda Vargas,** que siempre con su amor incondicional, está siempre a mi lado y siempre tuvo paciencia conmigo.

A mis hermanos; **Raúl**, **Yonatan**, **Rosmery**, **Jhoel** y **Jairo** que siempre confiaron en mí y me apoyaron incondicionalmente.

A mis docentes y compañeros de la universidad, porque me acompañaron en esta etapa de mi vida, por aquellos momentos inolvidables.

AGRADECIMIENTOS

Me gustaría que estas líneas sirvieran para expresar mis agradecimientos, en especial a Dios por permitirme llegar donde estoy.

Hago extensa esta gratitud a mis padres por el apoyo incondicional que me brindan, de los cuales estaré eternamente agradecido y bendecido por su apoyo.

Mis más sinceros agradecimientos a la Escuela Académico Profesional de Ingeniería Informática y Sistemas por la labor que hizo en fórmame profesionalmente.

Un agradecimiento en especial a los docentes de la EAPIIS por el conocimiento que compartieron conmigo.

Agradezco a mis compañeros de la universidad porque me acompañaron en esta etapa de mi vida, por aquellos momentos inolvidables.

INDICE

INDICE DE TABLAS

INDICE DE GRÁFICOS

INDICE DE FIGURAS

RESUMEN

El principal objetivo del trabajo ha sido el desarrollo de un compilador de pseudocódigo en español el cual facilite el aprendizaje de modularidad. El compilador de pseudocódigo está basado en los principales problemas de aprendizaje de la lógica de programación, la misma que se imparte dentro de la asignatura de Algorítmica I en la Escuela Académico Profesional de Ingeniería Informática y Sistemas; Se definió que dicha herramienta de aprendizaje consiste en un lenguaje de programación sencillo, junto a un entorno de desarrollo que proporcionará herramientas para la creación, compilación y ejecución de pseudocódigo. De manera que los estudiantes conozcan y adquieran habilidades básicas para el diseño de programas de computadora.

Existe la herramienta de desarrollo de pseudocódigo en español llamado PSeInt; el cual no cumple con la sintaxis necesaria para el aprendizaje de modularidad; es decir la capacidad de poder crear tanto funciones como procedimientos que permita dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable por los alumnos de la asignatura de Algorítmica I.

El tipo de investigación es aplicada con el nivel explicativo, utilizando la metodología XP para el desarrollo del software y C# para la programación.

La muestra es de 44 alumnos de la Asignatura de Algorítmica I grupo A, dividido en dos grupos independientes, el primer grupo de 16 alumnos del Semestre Académico 2015-I sin aplicar el compilador de pseudocódigo y el segundo grupo de 28 alumnos del Semestre Académico 2015-II aplicando el compilador de pseudocódigo; Los resultados demostrados mediante la prueba de hipótesis t de Student, con un nivel de significancia del 5%, donde : Tc=3.9060 > Tt=1.6820 por lo tanto se rechaza la hipótesis nula y se acepta la hipótesis alterna, por lo que se confirma que el compilador de pseudocódigo si contribuyó a la capacidad procedimental en el desarrollo de modularidad. En términos porcentuales tenemos un 71.43% de alumnos aprobados haciendo uso del compilador de pseudocódigo respecto al 50% de alumnos aprobados sin el compilador de pseudocódigo.

ABSTRACT

The main objective of the work was the development of a compiler pseudocode in Spanish which facilitates learning of modularity. The pseudo compiler is based on the main problems of learning programming logic, the same as is taught within the subject of Algorithmics I at the Academic Professional School of Computer Engineering and Systems; It was determined that this learning tool is a simple programming language, along with a development environment that provides tools for creating, compiling and running pseudocode. So that students know and acquire basic design computer software skills.

There is a development tool called PSeInt pseudocode in Spanish; which does not meet the required syntax for learning modularity; ie the ability to create both functions and procedures which divide a program into modules or subroutines in order to make it more readable and manageable by students of the subject of Algorithmics I.

The research is applied with the explanatory level, using the XP methodology for software development and C # for programming.

The sample is 44 students of the subject of Algorithmics I group A, divided into two separate groups, the first group of 16 students of the Academic Semester 2015-I without applying the compiler pseudocode and the second group of 28 students of the Academic Semester 2015-II applying pseudocode compiler; The results demonstrated by testing hypotheses Student t with a significance level of 5%, where: Tc = 3.9060> Tt = 1.6820 therefore the null hypothesis is rejected and the alternative hypothesis is accepted, so it is confirmed the compiler pseudocode if procedural capacity contributed to the development of modularity. In percentage terms we have a 71.43% pass compiler students using pseudocode about 50% of students approved without the compiler pseudocode.

INTRODUCCIÓN

El presente documento expone el proceso de desarrollo de un compilador, para el aprendizaje de modularidad mediante pseudocódigo en español. Una aplicación informática destinada a ser utilizada a manera de herramienta didáctica en los estudiantes de la asignatura de Algorítmica I, en la Escuela Académico Profesional de Ingeniería Informática y Sistemas de la Universidad Nacional Micaela Bastidas de Apurímac.

En el primer capítulo se expone el contexto sobre el que se desarrolló la tesis y se explica la situación problemática que se propone solucionar, así como las razones por las que se debe solucionar.

En el segundo capítulo se plantean los objetivos que se pretenden alcanzar. Se expone brevemente el objetivo general, seguido los objetivos específicos que lo complementan.

En el tercer capítulo se presenta una breve reseña de los antecedentes de la investigación, explicando los logros alcanzados y las dificultades encontradas. También se incluye los aspectos teóricos que se encuentran detrás del diseño y construcción de compiladores, explicando su estructura y componentes principales.

En el cuarto capítulo se plantea las hipótesis del informe de investigación así como la definición operacional de variables.

En el quinto capítulo se plantean las estrategias metodológicas utilizadas durante el desarrollo de la investigación.

En el sexto capítulo se detallan los resultados obtenidos a lo largo de la investigación. Para cada uno de ellos se exponen las técnicas utilizadas para su construcción, las estrategias escogidas para la implementación de sus componentes.

Por último se detallan las conclusiones y recomendaciones así como la bibliografía y anexos.

CAPÍTULO I
PLANTEAMIENTO DEL PROBLEMA

1.1. Definición y formulación del problema

Un Compilador se ha convertido en la actualidad en una herramienta que permite traducir un lenguaje de entrada (el lenguaje fuente) a un lenguaje salida (lenguaje objeto). Una parte importante del proceso de traducción es que el compilador avise de la presencia de errores al usuario en el programa fuente. De esta manera un programador puede diseñar un programa que permita definir un lenguaje mucho más cercano a cómo piensa un ser humano, para luego compilarlo a un programa más manejable por una computadora.

Algunas de las áreas de estudio más importantes en las que se apoya la Escuela Académico Profesional de Ingeniera informática y Sistemas de la Universidad Nacional Micaela Bastidas de Apurímac, se encuentran enfocadas en los fundamentos de la computación y el desarrollo de programas computacionales. Estas áreas de estudio comprenden asignaturas orientadas a la enseñanza del pensamiento abstracto y del razonamiento lógico-matemático a sus estudiantes, las cuales les permitan consolidar las bases sobre las que se fundamentará el conocimiento teórico.

El curso de Algorítmica I es fundamental para el inicio de la programación dentro de la carrera, ya que tiene como objetivo que sus estudiantes sepan identificar y utilizar las etapas del diseño de programas, conozcan y distingan los paradigmas de programación y adquieran habilidades básicas para el diseño de programas de computadora.

Al iniciarse en el mundo de programación, la pregunta clásica es: ¿Qué lenguaje de programación utilizar para aprender a programar?, la experiencia demuestra que mientras que no se domine los fundamentos de programación mediante algoritmo o pseudocódigos no será satisfactorio el aprendizaje, en este sentido hoy en día se hace uso de los pseudocódigo y diagramas para entender la lógica de programación. Existen dificultades en encontrar herramientas de desarrollo capaces de soportar este sistema de programación en pseudocódigo. Esto limita que la enseñanza pueda hacerse de manera más sencilla y eficaz. El problema radica en que no todos los

estudiantes de la asignatura de Algorítmica I, comprenden el lenguaje de los diagramas de flujos, puesto que no están familiarizados con el estudio de los mismos y mucho menos con los llamados lenguajes de programación de alto nivel.

Del mismo modo existe el intérprete de pseudocódigo en español llamado PSeInt; el cual no cumple con la sintaxis necesaria para el aprendizaje de modularidad; es decir la capacidad de poder crear tanto funciones como procedimientos que permita dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable por los alumnos de la asignatura de Algorítmica I.

Mencionada la problemática, la presente investigación pretende desarrollar un compilador de pseudocódigo el cual facilite el aprendizaje de la lógica de programación de los algoritmos y en particular el tema de modularidad de pseudocódigos, el cual permita la adquisición de un estilo correcto de programación en los alumnos de la asignatura de Algorítmica I. En ese entender se plantea las siguientes interrogantes:

Problema general

¿En cuánto facilita el compilador de pseudocódigo el aprendizaje en el desarrollo de modularidad, UNAMBA - 2015?

Problemas específicos

¿En cuánto facilita el compilador de pseudocódigo el aprendizaje en el desarrollo de procedimientos, UNAMBA - 2015?

¿En cuánto facilita el compilador de pseudocódigo el aprendizaje en el desarrollo de funciones, UNAMBA - 2015?

1.2. Justificación e importancia de la investigación

Dentro de la variada gama de aspectos que abarca la informática, en la actualidad cobra cada vez mayor importancia el desarrollo de software dirigidos al desarrollo de algoritmos mediante el uso de pseudocódigo. Se seleccionó el pseudocódigo porque es sencillo de usar ya que combina sentencias del lenguaje español y sentencias similares a las usadas por lenguajes de programación sin necesariamente seguir la estructura definida por estos, evitando la necesidad de codificar el algoritmo en un lenguaje de programación de alto nivel para poder validar los resultados.

De este modo no se necesitó seguir la sintaxis definida por algún lenguaje de programación para poder diseñar una solución verificable.

El propósito de la tesis fue de poder desarrollar un compilador de pseudocódigo y la estructura de sus elementos desde el analizador léxico, sintáctico y semántico para la generación del lenguaje objeto; la cual facilite el aprendizaje y el desarrollo modular de pseudocódigos permitiendo crear tanto funciones como procedimientos y dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

1.3. Limitaciones

El presente desarrollo del compilador de pseudocódigo se limitó a sistemas operativos Windows XP y sus versiones superiores, basándose en requisitos mínimos para ejecutar.

No se implementó la herramienta de generador de diagramas de flujo debido a que no se cuenta con una librería grafica para la representación de algoritmo.

1.4. Delimitación

1.4.1. Delimitación espacial

La investigación se desarrolló en la Universidad Nacional Micaela Bastidas de Apurímac, en la Escuela Académico Profesional de Ingeniería Informática y Sistemas dentro del curso de Algorítmica I del grupo A.

1.4.2. Delimitación temporal

El presente estudio de investigación se realizó en los Semestre Académicos 2015 – I y 2015 - II de la Universidad Nacional Micaela Bastidas de Apurímac.

1.5. Objetivos

1.5.1. Objetivo general

Desarrollar un compilador de pseudocódigo que facilite el aprendizaje en el desarrollo de modularidad, UNAMBA - 2015.

1.5.2. Objetivo especifico

- Facilitar el aprendizaje en el desarrollo de procedimientos mediante el compilador de pseudocódigo, UNAMBA - 2015.
- Facilitar el aprendizaje en el desarrollo de funciones mediante el compilador de pseudocódigo, UNAMBA - 2015.

CAPÍTULO II

MARCO REFERENCIAL

2.1. Antecedentes de la investigación

2.1.1. En el Perú

a) En el año 2013, JARA LOAYZA Juan Carlos, realizó la tesis para optar el título de Ingeniero Informático, que lleva como título "**Entorno de Desarrollo para la Ejecución y Traducción de Pseudocódigo**", de la Pontificia Universidad Católica del Perú – Perú.

En el trabajo de tesis se describe los problemas asociados al proceso de aprendizaje y enseñanza donde los alumnos invierten poco tiempo en el diseño del código, le brindan más importancia al lenguaje de programación que al proceso de construir un programa tomando un rol pasivo en el proceso de aprendizaje. Generalmente el aprendizaje se enfoca más en la sintaxis del lenguaje de programación que en el diseño del algoritmo.

En la misma tesis se planteó como objetivo desarrollar un entorno de desarrollo y un intérprete de pseudocódigo que genere y ejecute código en Visual Basic for Applications(VBA), como herramienta de apoyo al diseño, codificación y ejecución de un algoritmo. Y los objetivos específicos plantean definir la sintaxis del pseudocódigo para la representación del diseño del algoritmo, permitir la ejecución de algoritmos que sean diseñados bajo la estructura y sintaxis del pseudocódigo para su verificación. Permitir la codificación de pseudocódigo a código a lenguaje VBA, Python, Ruby, C++ y Java, por medio de traducción a código de un lenguaje a otro.

Las conclusiones que se obtuvieron fueron que el intérprete debe tener tipos de datos genéricos como Double o String, porque permite flexibilidad en las operaciones cuando aún no se conoce el tipo de dato. Es importante tener una buena representación de código intermedio (tabla de código y símbolos) esto permitió un sencillo recorrido para la ejecución, y además dependiendo de los tokens añadidos, la traducción a diferentes lenguajes de programación del pseudocódigo.

2.1.2. En el Exterior

a) En el año 2008, VEGA CASTRO Rafael Aníbal, realizó la tesis para optar el título de Ingeniero de Sistemas, que lleva como título "**Compilador de Pseudocódigo como Herramienta para el Aprendizaje en la Construcción de Algoritmos**", de la Universidad del Norte – Colombia.

En la tesis el principal problema que describe, que cuando los estudiantes se encuentran por primera vez frente a un algoritmo, se toma determinado tiempo para captar o entender cómo funciona este nuevo concepto. Los estudiantes no puedan realizar la comprobación de sus algoritmos en un software en donde la estructura del código que ellos ingresan sea exactamente la misma que los algoritmos estudiados en clase. El objetivo de la tesis fue desarrollar un compilador de código fuente en pseudocódigo para la construcción de algoritmos. Y los objetivos específicos plantean desarrollar un portal web de compilación de código fuente en pseudocódigo para la construcción de algoritmos, desarrollar un IDE (integrated development environment o entorno integrado de desarrollo) stand alone de compilación de código fuente en pseudocódigo para la construcción de algoritmos.

Las conclusiones que se obtuvieron fueron; para realizar un compilador no es necesario iniciar desde cero sino que se pueden tomar un conjunto de herramientas e integrarlas de cierto modo que cada una cumpla una función específica dentro del compilador para así poder lograr el objetivo final. El logro más interesante, es haber podido compilar el código fuente en pseudocódigo de manera remota por medio de una página Web.

b) En el año 2011, CANALES MARTÍNEZ Isaac Andrés y RUIZ TEJEIDA Michel, realizaron la tesis para optar el título de Ingeniero Informático, que lleva como título "**Desarrollo de un Compilador para Pseudocódigo en Lenguaje Español**", Instituto Politécnico Nacional – México.

En el trabajo de tesis se describe los problemas asociados al proceso de desarrollo de algoritmos, donde no es una buena práctica programar mientras se busca el algoritmo que da solución al problema, sino que es recomendable utilizar herramientas de apoyo tales como diagramas de flujo

o pseudocódigo para plantear de forma genérica los pasos a seguir, y posteriormente traducir a instrucciones particulares del lenguaje a utilizar. El objetivo del trabajo es mostrar el desarrollo de una herramienta capaz de procesar pseudocódigo, con reglas de sintaxis y palabras reservadas, para generar la transformación de dicho pseudocódigo a código Java, el cual se puede compilar y ejecutar. La herramienta desarrollada está orientada a ser usada por personas que están aprendiendo a programar, lo anterior se debe a que facilita la generación de código Java a partir de pseudocódigo en español.

Las conclusiones a las que llegaron fueron que el compilador es una herramienta diseñada para el estudiante y puede servir de guía al profesor, el desarrollo del compilador se inició y concluyó en Java el cual permitió su uso en cualquier sistema operativo que soporte la máquina virtual. Además disminuyo en gran medida que los aprendices de programación tengan que preocuparse por los tipos de datos, palabras en inglés, puntos y coma, y algunos otros detalles que el compilador de un lenguaje de alto nivel exige para la correcta compilación del código fuente.

c) En el año 2008, PÁEZ PÉREZ Liliam Paola y VÁSQUEZ ACERO John Henry, realizaron el trabajo de grado para optar el título de Ingeniero de Sistemas, que lleva como título "**PsiCoder: Software Educativo para Facilitar el Proceso de Enseñanza – Aprendizaje en un Curso Básico de Programación**", Pontificia Universidad Javeriana – Colombia.

En el trabajo de tesis se describe los problemas asociados a la impartición de la asignatura de programación sin acceso a herramientas que ayuden a afianzar los conocimientos de una manera más lúdica, así como considerar en el aspecto pedagógico los estilos de enseñanza y aprendizaje.

El bajo índice de aprobación de los estudiantes en los cursos de programación, deserción estudiantil, tanto de la asignatura durante un periodo académico, como el retiro total de la Carrera.

En la misma tesis se planteó como objetivo realizar una herramienta de software, como apoyo didáctico, que facilite la enseñanza y el aprendizaje de los conceptos básicos en la primera asignatura del área de programación en las carreras de ingeniería. Y los objetivos específicos fueron definir el estado del arte de las herramientas que apoyan los conceptos tratados en la asignatura de pensamiento algorítmico, diseñar y desarrollar una herramienta de software que funcione en los sistemas operativos windows y linux, capaz de ejecutar algoritmos en pseudocódigo.

Las conclusiones a la que llegaron fueron que la herramienta PsiCoder se basó en todo momento en el pro del aprendizaje de los estudiantes de la programación de computadores, y en el apoyo metodológico y pedagógico hacia los profesores y tutores de estos cursos. Esto permitió llevar hacia un grupo de estudiantes y profesores una herramienta madura y pertinente para su uso en la familiarización de los conceptos básicos en la programación, sustentando estas afirmaciones bajo los altos índices de aceptación alcanzados en la realización de la prueba piloto de usabilidad.

2.2. Marco teórico

2.2.1. Algoritmo

Un algoritmo es un método para resolver un problema. Aunque la popularización del término ha llegado con el advenimiento de la era informática, algoritmo proviene de Mohammed al-KhoWarizmi, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido es la palabra algoritmos que se derivó posteriormente en algoritmo. (Luís Joyanes Aguilar, 2008).

En la vida cotidiana se emplean algoritmos, frecuentemente para resolver problemas. Algunos ejemplos son los manuales de usuario, que muestran algoritmos para usar un aparato, o las instrucciones que recibe un trabajador por parte de su patrón.

Algunos ejemplos en matemáticas son: el algoritmo de la división para calcular el cociente de dos números, el algoritmo de Euclides para obtener el máximo común divisor de dos enteros positivos, o el método de Gauss para resolver un sistema lineal de ecuaciones. (Gálvez Rojas & Mora Mata, 2005).

2.2.1.1. Características principales y definición formal

En general, no existe ningún consenso definitivo en cuanto a la definición formal de algoritmo. Muchos autores lo señalan como listas de instrucciones para resolver un problema abstracto, es decir, que un número finito de pasos convierten los datos de un problema (entrada) en una solución (salida). Sin embargo cabe notar que algunos algoritmos no necesariamente tienen que terminar o resolver un problema en particular. Por ejemplo: una versión modificada de la criba de Eratóstenes que nunca termine de calcular números primos no deja de ser un algoritmo.

A lo largo de la historia varios autores han tratado de definir formalmente a los algoritmos utilizando modelos matemáticos. Sin embargo estos modelos están sujetos a un tipo particular de datos como son números, símbolos o gráficas mientras que los algoritmos funcionan sobre una extensa cantidad de estructuras de datos. En general, la parte común en todas las definiciones se puede resumir en las siguientes tres propiedades: (Gálvez Rojas & Mora Mata, 2005).

Tiempo secuencial. Un algoritmo funciona paso a paso, definiendo así una secuencia de estados computacionales por cada entrada válida (la entrada son los datos que se le suministran al algoritmo antes de comenzar).

Estado abstracto. Cada estado computacional puede ser descrito formalmente utilizando una estructura de primer orden y cada algoritmo es independiente de su implementación de manera que en un algoritmo y las estructuras de primer orden sean invariantes.

Exploración acotada. La transición de un estado al siguiente queda completamente determinada por una descripción fija y finita; es decir, entre cada estado y el siguiente solamente se puede tomar en cuenta una cantidad fija y limitada de términos del estado actual.

2.2.1.2. Medios de expresión de un algoritmo

Los algoritmos pueden ser expresados de muchas maneras, incluyendo al lenguaje natural, diagramas de flujo, pseudo-códigos, lenguajes de programación entre otros. Las descripciones en lenguaje natural tienden a ser ambiguas y extensas.

El usar pseudocódigo y diagramas de flujo evita muchas ambigüedades del lenguaje natural. Dichas expresiones son formas más estructuradas para representar algoritmos; no obstante, se mantienen independientes de un lenguaje de programación específico.

La descripción de un algoritmo usualmente se hace en tres niveles: (Brassard, Gilles; Bratley, Paul, 1997).

- **Descripción de alto nivel.** Se establece el problema, se selecciona un modelo matemático y se explica el algoritmo de manera verbal, posiblemente con ilustraciones y omitiendo detalles.
- **Descripción formal.** Se usan pseudocódigo para describir la secuencia de pasos que encuentran la solución.
- **Implementación.** Se muestra el algoritmo expresado en un lenguaje de programación específico o algún objeto capaz de llevar a cabo instrucciones.

2.2.1.3. Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son: (Brassard, Gilles; Bratley, Paul, 1997).

- Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
- Un algoritmo debe estar definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

2.2.1.4. Escritura de Algoritmos

Se emplea un lenguaje natural describiendo paso a paso el algoritmo en cuestión. En la realización de algoritmos aplicar sus tres características: preciso, definido y finito.

Por ejemplo algoritmo para conocer si el promedio de un alumno es aprobatorio o no teniendo como referencia que alumno cursa 5 materias y además que el promedio mínimo aprobatorio es 11. (Luís Joyanes Aguilar, 2008).

1. **Inicio**
2. Solicitar las cinco calificaciones del alumno
3. Sumar las cinco calificaciones del alumno
4. El resultado del paso 3 dividirlo entre cinco
5. Si el resultado del paso 4 es mayor o igual a 11 entonces

   5.1 Visualizar Alumno aprobado

   si_no

   5.2 Visualizar Alumno reprobado

   fin_si
6. **Fin**

En el algoritmo anterior se dio solución al planteamiento básico del cálculo del promedio de un alumno. Se observa que los pasos del algoritmo tienen un número que conforme se va describiendo la secuencia, ese número se va incrementando. Es importante destacar que todo algoritmo es finito, es decir, así como tiene un inicio debe tener un fin, lo que se observa en los pasos 1 y 6. (Luís Joyanes Aguilar, 2008).

2.2.1.5. Representación gráfica de los algoritmos

Para la representación gráfica del algoritmo debe emplearse un método que sea independiente del lenguaje de programación elegido.

Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica o numéricamente, de modo que las sucesivas acciones no dependan de la sintaxis de ningún lenguaje de programación, sino que la descripción se emplee para su transportación en un programa. (Luís Joyanes Aguilar, 2008).

Los métodos usuales para representar un algoritmo son:

1.- Diagrama de flujo.

2.- Diagrama N-S (Nassi-Schneiderman).

3.- Lenguaje de especificación de algoritmos: pseudocódigo.

4.- Lenguaje español, inglés.

5.- Fórmulas.

2.2.1.6. Pseudocódigo

El pseudocódigo (falso lenguaje, por el prefijo pseudo que significa falso) es una descripción de alto nivel de un algoritmo que emplea una mezcla de lenguaje natural con algunas convenciones sintácticas propias de lenguajes de programación, como asignaciones, ciclos y condicionales, aunque no está regido por ningún estándar.

Es utilizado para describir algoritmos en libros y publicaciones científicas, y como producto intermedio durante el desarrollo de un algoritmo, como los diagramas de flujo, aunque presentan una ventaja importante sobre estos, y es que los algoritmos descritos en pseudocódigo requieren menos espacio para representar instrucciones complejas. (Gálvez Rojas & Mora Mata, 2005).

El pseudocódigo está pensado para facilitar a las personas el entendimiento de un algoritmo, y por lo tanto puede omitir detalles irrelevantes que son necesarios en una implementación. Programadores diferentes suelen utilizar convenciones distintas, que pueden estar basadas en la sintaxis de lenguajes de programación concretos. Sin embargo, el pseudocódigo en general es comprensible sin necesidad de conocer o utilizar un entorno de programación específico, y es a la vez suficientemente estructurado para que su implementación se pueda hacer directamente a partir de él. (Gálvez Rojas & Mora Mata, 2005).

Ventajas del uso de pseudocódigo reside en:

- Su uso en la planificación de un programa; permitiendo que el programador se pueda concentrar en la lógica y en las estructuras de control y no tenga que preocuparse, por ahora de detalles acerca de las reglas sintácticas y semánticas de un lenguaje especifico. Consiguientemente es más fácil de modificar, en el caso de que se descubra errores o anomalías en la lógica del algoritmo.
- Aunque el pseudocódigo es independiente del lenguaje de alto nivel que vaya a usarse, un algoritmo expresado en pseudocódigo puede ser traducido más fácilmente a muchos de ellos. (Gálvez Rojas & Mora Mata, 2005).

2.2.2. Desarrollo de modularidad

A medida que avanza el tiempo, los problemas que se espera que un computador resuelva se vuelven más y más complejos. Así, es en realidad muy extraño que se pueda diseñar un algoritmo simple que resuelva un problema real hoy en día, por lo que es necesario plantearse algún tipo de subdivisión que haga el problema abordable, en sub problemas manejables. En este tema se presentan las herramientas de programación que permiten resolver problemas complejos mediante su descomposición en otros más simples (Baltasar García Perez-Schofield, 2012).

Un algoritmo que resolviera un problema complejo, contendría cientos o miles de líneas de código en su interior. Esto es inabarcable para cualquier programador, por lo que se utiliza el concepto de procedimientos y funciones para subdidivir el problema en partes.

La idea es que cada una de estas partes contenga un conjunto de instrucciones que permita la ejecución de algún proceso determinado y lógico desde el punto de vista humano. (Baltasar García Perez-Schofield, 2012).

2.2.2.1. Funciones

Una función es una un conjunto de instrucciones, con un nombre asociado, que cumple las siguientes características: (Baltasar García Perez-Schofield, 2012).

- Tiene uno o más parámetros de entrada.
- Tiene un parámetro exclusivamente de salida y de tipo simple.
- Todos los valores de entrada son necesarios y suficientes para determinar el valor de salida.

Su sintaxis es la siguiente:

```
FUNCION nombre_funcion(parámetros_entrada) : Tipo_salida
        CONSTANTES
        TIPOS
        VARIABLES
        INICIO
                Instrucción 1
                Instrucción 2
                ....
                RETORNA (expresión){ de Tipo_de_salida }
FIN_FUNCION
```

2.2.2.2. Procedimientos

Son conjuntos de instrucciones con un nombre asociado, al igual que las funciones, pero no devuelven ningún valor. Los parámetros pueden ser de entrada, salida o de entrada / salida. (Baltasar García Perez-Schofield, 2012).

Su sintaxis es la siguiente:

```
PROCEDIMIENTO nombre_procedimiento(parámetros_entrada)
        CONSTANTES
        TIPOS
        VARIABLES
        INICIO
                Instrucción 1
                Instrucción 2
                ....
FIN_PROCEDIMIENTO
```

2.2.3. Traductor

Un traductor se define como un programa que traduce o convierte desde algún texto o programa escrito en un lenguaje fuente hasta un texto, ejecución o programa equivalente, escrito en un lenguaje destino, produciendo si cabe mensaje de error.

Los traductores engloban tanto a los compiladores (en los que el lenguaje destino suele ser código máquina) como a los intérpretes (en los que el lenguaje destino está constituido por las acciones atómicas que puede ejecutar el intérprete). (Sergio Gálvez R. y Miguel A. Mora M, 2005).

2.2.3.1. Estructura de un traductor

Un traductor divide su labor en dos etapas: una que analiza la entrada y genera estructuras intermedias y otra que sintetiza la salida a partir de dichas estructuras. (Galvez Rojas & Mora Mata, 2005).

Básicamente los objetivos de la etapa de Análisis son:

a) Controlar la corrección del programa fuente, y
b) Generar las estructuras necesarias para comenzar la etapa de síntesis.

Para llevar esto a cabo, la etapa de análisis consta de las siguientes fases:

Análisis lexicográfico. En esta fase se divide el programa fuente en los componentes léxicos del lenguaje a compilar. Cada componente léxico es una subsecuencia de caracteres del programa fuente, y pertenece a una categoría gramatical: números, identificadores de usuario (variables, constantes, tipos, nombres de procedimientos, etc.), palabras reservadas, signos de puntuación, etc. (Galvez Rojas & Mora Mata, 2005).

Análisis sintáctico. En esta fase se comprueba que la estructura de los componentes léxicos sea correcta según las reglas gramaticales del lenguaje que se compila. (Galvez Rojas & Mora Mata, 2005).

Análisis semántico. En esta fase se comprueba que el programa fuente respete las directrices del lenguaje que se compila (todo lo relacionado con el significado): chequeo de tipos, rangos de valores, existencia de variables, etc.

Cualquiera de estas tres fases puede emitir mensajes de error derivados de fallos cometidos por el programador en la redacción de los textos fuente. Mientras más errores controlen el compilador, menos problemas darán un programa en tiempo de ejecución. Por ejemplo, el lenguaje C no controla los límites de un array, lo que provoca que en tiempo de ejecución puedan producirse comportamientos del programa de difícil explicación. (Galvez Rojas & Mora Mata, 2005).

2.2.4. Compilador

Es aquel traductor que tiene como entrada una sentencia en lenguaje formal y como salida tiene un fichero ejecutable, es decir, realiza una traducción de un código de alto nivel a código máquina. También se entiende por compilador aquel programa que pasando por diferentes fases de traducción proporciona o no un fichero objeto en lugar del ejecutable final. (Gálvez Rojas & Mora Mata, 2005).

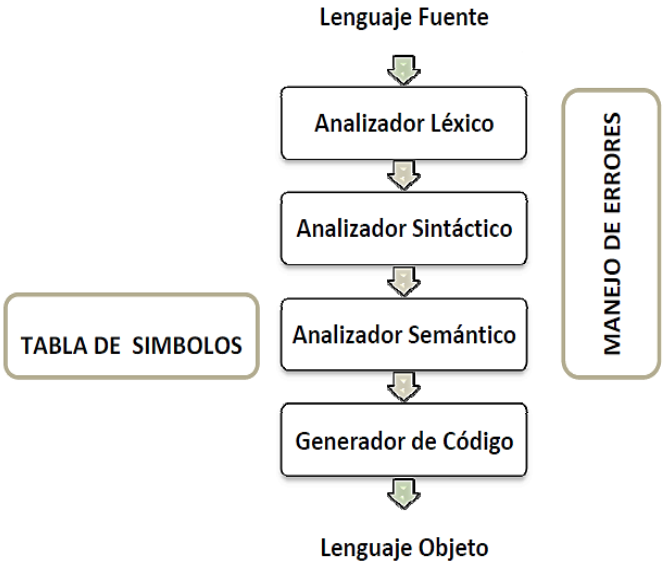


Figura N° 1: Fases de un compilador
Fuente: Adaptado de Sergio Gálvez R. y Miguel A. Mora M

2.2.4.1. Estructura de un compilador

Analizador Léxico:

Es la primera fase del compilador que se encarga de leer el código fuente y procesarlo, también es conocido como escaneo por su nombre en inglés (scanner). Durante este proceso se realizan operaciones básicas de manejo de cadenas de acuerdo a ciertas reglas del lenguaje, estas reglas las conocemos en teoría computacional como expresiones regulares. (Sergio Gálvez R. y Miguel A. Mora M, 2005).

Analizador Sintáctico

La siguiente fase de compilación es el analizador sintáctico (parser), en donde se analiza la estructura gramatical del lenguaje fuente, estas reglas son representadas por las gramáticas libres del contexto y su escaneo con los árboles sintácticos. (Sergio Gálvez R. y Miguel A. Mora M, 2005).

Analizador Semántico

En la tercera etapa aún se analiza el código fuente para verificar las reglas semánticas, estas reglas están representadas por la correspondencia de los tipos de datos que se manejen en el lenguaje. (Sergio Gálvez R. y Miguel A. Mora M, 2005).

Generador de Código Intermedio

Una vez analizado el código fuente y sin haber encontrado algún error durante las fases anteriores, el compilador genera un código intermedio para prepararlo al código destino. (Sergio Gálvez R. y Miguel A. Mora M, 2005).

Generador de Código

La quinta etapa del compilador también conocida como generador de código o back-end, es el proceso encargado de traducir el lenguaje intermedio a un código máquina, el cual dependerá de la arquitectura de la misma. (Sergio Gálvez R. y Miguel A. Mora M, 2005)

Optimizador de Código

El último proceso de un compilador es opcional, ya que la optimización de código depende de la arquitectura de la máquina, si ésta acepta paralelismo y que tipo de paralelismo, además de que no es posible saber a ciencia cierta si un código optimizado es mejor que el código ingresado por el programador (Ruiz Catalán, 2010).

2.2.5. Intérprete

Es como un compilador, sólo que la salida es una ejecución. El programa de entrada se reconoce y ejecuta a la vez. No se produce un resultado físico (código máquina) sino lógico (una ejecución). Hay lenguajes que sólo pueden ser interpretados, como por ejemplo LISP (LISt Processing), algunas versiones de BASIC (Beginner‟s All-purpose Symbolic Instruction Code), etc.

Su principal ventaja es que permiten una fácil depuración. Entre los inconvenientes podemos citar, en primer lugar, la lentitud de ejecución, ya que al ejecutar a la vez que se traduce no puede aplicarse un alto grado de optimización; por ejemplo, si el programa entra en un bucle y la optimización no está muy afinada, las mismas instrucciones se interpretarán y ejecutarán una y otra vez, enlenteciendo la ejecución del programa.

Además de que la traducción optimiza el programa acercándolo a la máquina, los lenguajes interpretados tienen la característica de que permiten construir programas que se pueden modificar a sí mismos. (Galvez Rojas & Mora Mata, 2005).

2.2.6. Notación de Backus-Naur

La notación de Backus-Naur, también conocida por sus denominaciones inglesas Backus-Naur form (BNF), Backus-Naur formalism o Backus normal form, es una metasintaxis usada para expresar gramáticas libres de contexto: es decir, una manera formal de describir lenguajes formales.

El BNF se utiliza extensamente como notación para las gramáticas de los lenguajes de programación de la computadora, de los sistemas de comando y de los protocolos de comunicación, así como una notación para representar partes de las gramáticas de la lengua natural (por ejemplo, el metro en la poesía de Venpa). La mayoría de los libros de textos para la teoría o la semántica del lenguaje de programación documentan el lenguaje de programación en BNF. (Knuth Donald E, 1964).

Algunas variantes, tales como la Augmented Backus-Naur form (ABNF) y la Extended Backus–Naur Form (EBNF), tienen su propia documentación.

2.2.6.1. Variantes

Hay muchas variantes y extensiones de BNF, posiblemente conteniendo algunos o todos los comodines de expresiones regulares como un "*" o "+". El Extended Backus-Naur form (EBNF) es una variante común. La notación de los corchetes "[ ]" fue introducida algunos años más tarde en la definición de PL/I de la IBM pero ahora se reconoce universal.

Las expresiones gramaticales de analizadores sintácticos construidas en BNF y las notaciones de expresión regular para formar una clase alternativa de la gramática formal, que es esencialmente analítica más que generativa en carácter.

Una especificación de BNF es un sistema de reglas de derivación, escrito como:

<simbolo> ::= <expresión con símbolos>

Donde <símbolo> es un no terminal, y la expresión consiste en secuencias de símbolos o secuencias separadas por la barra vertical, '|', indicando una opción, el conjunto es una posible substitución para el símbolo a la izquierda. Los símbolos que nunca aparecen en un lado izquierdo son terminales. (Knuth Donald E, 1964).

2.2.7. Metodología XP

La metodología XP se considera una metodología leve desarrollo de software. Esta es clasificada como un sistema de prácticas que la comunidad de desarrolladores de software viene evolucionando para resolver los problemas de entrega de software de calidad rápidamente, y poder alcanzar las necesidades de negocio que siempre cambian. Esta surgió a partir de ideas de Kent Beck y Ward Cunningham y que fue utilizada por primera vez en un proyecto piloto en marzo de 1996, del cual el propio Keck formaba parte. Lo de Extreme del nombre de la metodología se debe al hecho de que esta emplea al extremo, las buenas prácticas de la ingeniería del software.

La XP no se aplica a todos los tipos de proyectos, siendo más apropiado para los proyectos con equipos pequeños o medianos, de dos a doce personas. Sin embargo, algunos defienden su uso en grande proyectos, ya que el dividirlos en sub proyectos independientes. Los proyectos largos deben ser partidos en una secuencia de mini proyectos de auto contenido, una duración de una a tres semanas (José Rubén Laínez, 2015).

XP es un proceso de desarrollo de software apropiado para siguiente proyectos:

- Con requisitos que cambian con frecuencia
- Desarrollo de sistemas orientado a objetos
- Equipos pequeños
- Desarrollo incremental

Valores de la metodología XP

XP está definida por medio de valores, principios y prácticas. Los valores describen los objetos de largo plazo y definen criterios para obtener el éxito. Los valores son: Feedback, comunicación, simplicidad, coraje y respeto.

Cuando un cliente aprende el sistema que utiliza y evalúa sus necesidades, este genera un feedback para el equipo de desarrollo. Es decir, este realimenta al equipo con modificaciones en las necesidades que aun serán implementadas y eventualmente en aquellas que ya forman parte del software. El feedback es un mecanismo fundamental que permite al cliente conducir el desarrollo diariamente y garantizar que el equipo direccione sus atenciones para aquello que genera más valor. (José Rubén Laínez, 2015).

La comunicación entre el cliente y el equipo permite que todos los detalles del proyecto sean tratados con atención y agilidad que se merecen. La metodología XP busca asegurar que la comunicación ocurra de la forma más directa y eficaz posible. Siendo así, esta busca aproximar a todo los participantes del proyecto a un diálogo presencial. (José Rubén Laínez, 2015).

Prácticas de la metodología XP

La metodología XP tiene una docena de práctica derivadas de sus valores, esas prácticas definen su uso y son las siguientes: (José Rubén Laínez, 2015).

> **Cliente Presente:** Uno de los paradigmas del desarrollo del software tradicional es que el cliente no necesita, o incluso no debe estar presente durante el proceso de desarrollo. XP busca acabar con ese paradigma, haciendo que la presencia del cliente sea de vital importancia para el éxito del proyecto. El feedback que el cliente suministra es parte esencial de una iteración. Para que el cambio de feedback pueda ocurrir y el cliente pueda obtener el máximo de su valor del proyecto, es esencial que se participe activamente del proceso de desarrollo.

Juego de la planificación: En el inicio de cada iteración el cliente es invitado a escribir, a su manera las funcionalidades que desea en el sistema, en pequeñas tarjetas llamadas historias de usuario, esas es la cantidad más pequeña de información que este puede especificar. Esta práctica es conocida como el juego de planificación y asegura que este equipo trabaje en el que considere más importante para el cliente.

Programación en Par: En la programación en par, dos desarrolladores escogen una historia de usuario y se sientan en un único ordenador para codificar una determinada funcionalidad. El desarrollador con menos experiencia tiene como responsabilidad asumir el control del teclado y conducir activamente la programación del código fuente, mientras el otro con mayor experiencia inspecciona el código en busca de errores y defectos, cuestionando las decisiones y buscando estratégicamente las soluciones más simples para el código. Uno de los beneficios de esa práctica es la revisión constante del código, y también la diseminación del conocimiento entre los pares, lo que ayuda a la nivelación técnica de todo el equipo.

Código Colectivo: En la XP, el sistema no está segmentado en partes. Los desarrolladores tienen acceso a todas las partes del código y pueden modificar aquello que juzguen importante: el código es colectivo. Eso suministra mayor agilidad al proceso y crea más un mecanismo de revisión y de verificación del código, ya que aquello que es escrito por un par, acaba siendo manipulado por otro. Si alguna cosa está confusa en el código, este pasará por el refactoring.

Código Estandarizado: Para que todos los desarrolladores puedan manipular cualquier parte del software, cie forma más rápida, el equipo establece estándares de codificación, que sirven también para hacer el sistema más homogéneo, permitiendo que cualquier mantenimiento futuro sea efectuado más rápidamente.

Integración Continua: La práctica de la integración continua es la actividad de unir el trabajo realizado por un par de programadores, al código como uno todo. Después de terminar determinada actividad. El par debe probar y juntar su código a la versión más reciente del código

colectivo. Eso debe ser hecho varias veces al día, para sincronizar las actividades individuales.

Ritmo Sustentable: Esa práctica consiste en trabajar respetando los límites físicos y demostrando respeto por la individualidad. Para eso, la XP recomienda que la carga horaria de trabajo no pase de las 8 horas diarias y 40 horas semanales.

Stand Up Meeting: El equipo de desarrollo se reúne cada mañana para evaluar el trabajo que fue ejecutado el día anterior y priorizar aquello que será implementado el día que se inicia. Se trata de una reunión rápida que recibe el nombre de stand up meeting, que en inglés significa reunión en pie.

2.2.8. ISO 9241

2.2.8.1. Usabilidad

El grado en el cual un producto puede ser usado por unos usuarios específicos para alcanzar ciertas metas específicas con efectividad, eficiencia y satisfacción en un contexto de uso específico.

- Facilidad de Aprendizaje: Indica qué tan fácil es aprender la funcionalidad básica del sistema, como para ser capaz de realizar las tareas que desea realizar el usuario.
- Eficiencia: La eficiencia se determina por el número de transacciones por unidad de tiempo que el usuario puede realizar usando el sistema.
- Facilidad de recordad: Cuando un usuario ha utilizado un sistema tiempo atrás, y tiene la necesidad de utilizarlo, de nuevo la curva de aprendizaje debe ser significativamente menor que el caso del usuario que nunca haya utilizado dicho sistema.
- Manejo de errores: Indica cómo el sistema previene los errores que el usuario puede cometer mientras se encuentra operando el sistema.

- Satisfacción: Indica la impresión subjetiva que el operador del sistema obtiene del mismo. Para ello se utilizan cuestionarios, encuestas, entrevistas.
- Nivel de seguridad: La calidad no puede existir sin seguridad. Constituye un factor importante en la usabilidad de una aplicación, porque genera mayor confianza en los usuarios.

2.2.8.2. Ergonomía

Estudia a los puestos de trabajo en computación (PTC) desde dos dimensiones: la Ergonomía del Hardware y la Ergonomía del Software. La Ergonomía del Software es una disciplina que contribuye con sus métodos y conocimientos al estudio de la Interacción entre Persona – Ordenador. La Ergonomía del software es la disciplina que logra que esa interfaz sea usable.

2.2.8.3. ISO 9241 – 10: Principios para diálogos

Esta parte describe principios generales de ergonomía juzgados importantes para el diseño y evaluación de diálogos entre el usuario y los sistemas de información (adaptación a la tarea, carácter auto descriptivo, control por parte del usuario, conformidad con las expectativas del usuario, tolerancia a errores, aptitud a la individualización, facilidad de aprendizaje). Estos principios pueden ser aplicados durante la especificación, el desarrollo o la evaluación de software como línea directriz general, y son independientes de cualquier técnica de diálogo específico. En este documento, cada principio está acompañado de una descripción seguida de ejemplos de puesta en práctica.

2.2.8.4. ISO 9241 – 11: Guía de especificaciones y medidas de usabilidad

Esta parte define la usabilidad y explica cómo identificar la información a tomar en cuenta para especificar o evaluar la usabilidad, en términos de desempeño y satisfacción del usuario.

2.2.8.5. ISO 9241 – 12: Presentación de la información

Esta parte proporciona recomendaciones ergonómicas relativas a la presentación y a las propiedades particulares de la información presentada en pantallas de visualización. Las recomendaciones proporcionadas tienen como objetivo permitir al usuario ejecutar tareas de percepción de manera eficaz y satisfactoria.

2.2.8.6. ISO 9241 – 13: Guía del usuario

Esta parte proporciona recomendaciones relativas a la ayuda del usuario. Las recomendaciones presentadas en esta parte están relacionadas al prompt, el feedback, el estado del sistema, la gestión de errores y la ayuda en línea. Las recomendaciones presentadas en esta parte deberían facilitar la interacción de un usuario con un programa, favoreciendo el uso eficaz del programa, proporcionando a los usuarios un medio de gestión de errores y un asistente a los usuarios con niveles de conocimiento diferente.

2.2.9. Modelo Vista Controlador

Uno de los marcos de trabajo más conocido y ampliamente usado para el diseño de GUI es el marco Modelo - Vista-Controlador (MVC). El marco de trabajo MVC fue propuesto originalmente en la década de los 80 como una aproximación al diseño de GUI que permitió múltiples presentaciones de un objeto y estilos independientes de interacción con cada una de estas presentaciones. El marco MVC soporta la presentación de los datos de diferentes formas e interacciones independientes con cada una de estas presentaciones. Cuando los datos se modifican a través de una de las presentaciones. El resto de las presentaciones son actualizadas.

El "Modelo", que es la aplicación del dominio, no tiene conocimiento específico de la Vista, que es la Interfaz de Usuario, ni de los controladores, que son los manejadores de eventos ("listener").

La Vista (Interfaz del Usuario) tiene conocimiento de los controladores (los manejadores de eventos) porque los crea y especifica que tipo de evento, emitido por cada componente, debe de atender cada manejador. Sin embargo la vista no tiene conocimiento del Modelo.

Los Controladores tienen conocimiento del Modelo porque crean objetos del dominio al que mandan mensajes, derivados fundamentalmente de los eventos producidos en la interfaz de usuario: y tienen conocimiento de la Vista (Interfaz de Usuario) porque mandan mensajes a sus componentes para recibir o enviarles información. (Ian Sommerville, 2005)

2.2.10. **Aprendizaje**

El aprendizaje y la memoria son dos procesos íntimamente relacionados, por lo que es extremadamente difícil separar uno de otro. Gracias a ellos, adquirimos nuevos conocimientos, conductas y aptitudes, pero también son la base del crecimiento emocional, la adquisición de valores y actitudes e incluso de la formación de nuestra personalidad. Además, no sólo aprendemos cosas nuevas, sino que somos capaces de modificar lo aprendido para mejorarlo y adaptar de forma eficaz nuestra conducta ante diferentes situaciones y momentos.

El aprendizaje está presente a lo largo de toda la vida, no sólo en las etapas tempranas del desarrollo, en las cuales, por supuesto, es fundamental. Un niño aprende no sólo hechos y conocimiento acerca del mundo que le rodea, sino también de las personas que lo acompañan y de sí mismo. Por lo tanto, habrá aprendizajes que resulten más evidentes y observables que otras, de carácter más sutil. Además, los motivos por los que aprendemos pueden ser muy diferentes ya que en ocasiones nos guían recompensas externas, como por ejemplo el dinero, y en otras tenemos razones de carácter más interno, como la satisfacción personal.(Elena Muñoz y José Antonio Periáñez, 2012)

2.2.10.1. Concepto actual de aprendizaje y memoria

El aprendizaje puede definirse como la modificación relativamente estable y permanente de nuestra conducta o cognición como resultado de la experiencia. Las modificaciones debidas a la maduración o a estados transitorios o inducidos de un organismo, como por ejemplo el estrés o los cambios inducidos por fármacos, no son considerados aprendizaje, aunque pueden facilitar la aparición de nuevos aprendizajes.

Por su parte, la memoria constituye el proceso por el cual los nuevos conocimientos o sucesos son codificados, almacenados y, más tarde, recuperados. El concepto de memoria no es un concepto unitario; existen diferentes tipos o clases de memoria según se atienda a criterios temporales, categoriales, intencionales, etc.

Todas ellas han sido descritas a lo largo de los años a partir de la investigación básica y de la experiencia clínica. (Elena Muñoz y José Antonio Periáñez, 2012)

2.2.10.2. Fases del proceso de aprendizaje y memoria

Codificación

Implica el procesamiento, consciente o inconsciente, de la información a la que se atiende, con el fin de que sea almacenada posteriormente. Consiste en la transformación de los estímulos sensoriales en diferentes códigos de almacenamiento. La codificación constituye un proceso imprescindible para que la información sea almacenada, y puede producirse a partir de diferentes modalidades sensoriales, siendo más eficaz la codificación que se realiza basándose en más de una modalidad.

Almacenamiento o consolidación

En esta fase se crea y se mantiene un registro temporal o permanente de la información. El material almacenado posee en este momento una alta organización, lo que facilita el aumento en la cantidad de in formación que puede ser almacenada.

Posteriormente, la in formación almacenada puede perderse por diferentes motivos, tales como el olvido.

Recuperación

Hace referencia al acceso y evocación de la información almacenada a partir del cual se crea una representación consciente o se ejecuta un comportamiento aprendido. El proceso de recuperación puede llevarse a cabo de diferentes maneras.

2.3. Marco conceptual

a) Desarrollo de modularidad

Programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable. (Baltasar García Perez-Schofield, 2012).

b) Funciones

Es un conjunto de instrucciones, con un nombre asociado, que tiene uno o más parámetros de entrada y un parámetro exclusivamente de salida (Baltasar García Perez-Schofield, 2012).

c) Procedimientos

Son conjuntos de instrucciones con un nombre asociado, al igual que las funciones, pero no devuelven ningún valor (Baltasar García Perez-Schofield, 2012).

d) Compilador

Programa de computadora que produce un programa en lenguaje de máquina, de un programa fuente que generalmente está escrito por el programador en un lenguaje de alto nivel (Gálvez Rojas & Mora Mata, 2005).

e) Pseudocódigo

Es una descripción de un algoritmo de programación informático de alto nivel compacto e informal.

f) Herramienta

Permite reproducir la funcionalidad de una Aplicación informática.

g) Prueba unitaria

Es un tipo de prueba que se realiza de manera aislada a un componente de un sistema con el fin de verificar que produce los resultados esperados.

h) Algoritmo

Es un conjunto finito de instrucciones o pasos que sirven para ejecutar una tarea o resolver un problema.

i) Traductor

Programa que traduce o convierte desde algún texto o programa escrito en un lenguaje fuente hasta un texto, ejecución o programa equivalente.

j) Analizador léxico

Etapa que lee todo el código fuente y se agrupa y clasifica los caracteres en grupos con significados llamados lexemas, cada uno de esto produce un token el cual cuenta con un nombre y valor. (Sergio Gálvez R. y Miguel A. Mora M, 2005)

k) Analizador Sintáctico

Etapa que genera un árbol sintáctico a partir de los tokens, el cual sigue una estructura gramatical. Se comprueba que todo lexema sea válido y reconocible, y que la estructura de los lexemas cumpla con las reglas gramáticas del lenguaje. (Sergio Gálvez R. y Miguel A. Mora M, 2005)

l) Analizador Semántico

Etapa que se enfoca a revisar el significado del código. Puede evaluar en esta etapa la validez de las expresiones, chequeo de tipos, rangos de valores, existencia de variables, etc. (Sergio Gálvez R. y Miguel A. Mora M, 2005).

m) Token

Son unidades lógicas que genera el analizador léxico. Es el conjunto de cadenas de entrada que produce como salida el mismo componente léxico.

n) Patrón

Regla que describe la secuencia de caracteres que puede representar a un determinado componente léxico en los programas fuente.

CAPÍTULO III

HIPÓTESIS Y VARIABLES

3.1. Formulación de la hipótesis

3.1.1. Hipótesis general

El compilador de pseudocódigo facilitará el aprendizaje en el desarrollo de modularidad, UNAMBA - 2015.

3.1.2. Hipótesis especifica

a) El compilador de pseudocódigo facilitará el aprendizaje en el desarrollo de procedimientos, UNAMBA - 2015.

b) El compilador de pseudocódigo facilitará el aprendizaje en el desarrollo de funciones, UNAMBA - 2015.

3.2. Sistema de variables

Variable independiente = Compilador

Variable dependiente = Aprendizaje de modularidad

3.3. Definición operacional de variables

Tabla N° 1: Definición operacional de variables

VARIABLE	DIMENSIÓN	INDICADOR	INDICÉ / ESCALA
INDEPENDIENTE **Compilador,** Programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz de interpretar. (Gálvez Rojas & Mora Mata, 2005).	ISO/IEC 9241. (ISO 9241)	Usabilidad (ISO 9241)	- Bueno - Regular - Malo
		Ergonomía (ISO 9241)	- Bueno - Regular - Malo
DEPENDIENTE **Aprendizaje de modularidad.** Programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable. (Baltasar García Perez-Schofield, 2012).	Aprendizaje de modularidad	Aprendizaje en el desarrollo de procedimientos	- Notas (0-20)
		Aprendizaje en el desarrollo de funciones	- Notas (0-20)

Fuente: Elaboración propia

CAPÍTULO IV
METODOLOGÍA DE LA INVESTIGACIÓN

4.1.Tipo y nivel de investigación

El tipo de investigación que se realizó en el presente estudio de investigación, es una investigación aplicada por que persigue fines más directos e inmediatos. Tal es el caso, sobre el estudio que se propuso evaluar los efectos al aplicar el compilador de pseudocódigo como herramienta para facilitar el aprendizaje en el desarrollo de modularidad en la Escuela Académico Profesional de Ingeniería Informática y Sistemas.

El nivel de investigación es explicativo. Debido a que es importante explicar los efectos producidos del compilador de pseudocódigo como herramienta de aprendizaje en el desarrollo de modularidad en la Escuela Académico Profesional de Ingeniería Informática y Sistemas.

4.2. Método y diseño de investigación

El método usado en este trabajo de investigación es el método deductivo-inductivo. Este método permite describir, detallar, especificar todos los aspectos del desarrollo de un compilador de pseudocódigo en español; ya que permite responder a las preguntas planteadas.

El diseño de investigación que se realizó es cuasi experimental. Este diseño de investigación se ajusta más a la realidad de la problemática, puesto que es necesario observar los efectos en la variable dependiente aplicando la variable independiente; permitirá detallar las características y funcionalidades desarrolladas en el compilador de pseudocódigo.

4.3. Población y muestra de la investigación

Población:

La población de la investigación es el número de estudiantes de la Asignatura de Algorítmica I de los semestres académicos 2015-I y 2015-II de la Escuela Académico Profesional de Ingeniería Informática y Sistemas de la Universidad Nacional Micaela Bastidas de Apurímac.

N= 84 estudiantes

N= es la cantidad de estudiantes.

Muestra:

La muestra de la investigación se selecciona utilizando un muestro no probabilístico, muestreo intencional elegida por el investigador.

Para ello se eligió a los estudiantes de la asignatura de Algorítmica I grupo A, dividido en dos grupos independientes, el primer grupo de 16 alumnos del Semestre Académico 2015-I y el segundo grupo de 28 alumnos del Semestre Académico 2015-II, que corresponde al área de formación profesional obligatorio (especialidad).

n=44 alumnos de la asignatura de Algorítmica I del Grupo A.

Donde n es el tamaño de la muestra.

4.4. Procedimientos de la investigación

I Etapa: Desarrollo del compilador de pseudocódigo utilizando la metodología XP

En esta etapa se desarrolló el compilador de pseudocódigo utilizando la metodología XP, basada en un desarrollo de realimentación continua, especialmente usado para software con requisitos cambiantes. Esta característica de XP fue muy importante para el proyecto, ya que permitió que cada elemento del compilador (analizador sintáctico, optimizador de código, etc.) sea probado y con esto tener componentes cuyo funcionamiento se encuentran totalmente verificados.

- Planeación
- Diseño
- Codificación

II Etapa: Prueba y depuración de errores

En esta etapa se desarrolló las pruebas de errores. Se enfatiza mucho los aspectos relacionados con las pruebas, clasificándola en diferentes tipos y funcionalidades específicas.

- Pruebas unitarias.
- Pruebas de sistema.
- Cuando se encuentra un error.

III Etapa: Procesamiento de datos

En la etapa de procesamiento de los datos se aplicó una medición de las variables.

- Registro de la información sin la aplicación del compilador de pseudocódigo en la asignatura de Algorítmica I.
- Registro de la información con la aplicación del compilador de pseudocódigo en la asignatura de Algorítmica I.

IV Etapa: Tratamiento de datos

En esta etapa se realizó un tratamiento de los datos tomados sin la aplicación del compilador de pseudocódigo y con la aplicación del compilador de pseudocódigo en la asignatura de Algorítmica I.

- Comparación de los resultados.

V Etapa: Informe final

En esta etapa se realizó la redacción del informe final.

- Redacción del informe final.

4.5. Material de investigación

- Notación Backus Naur Form (BNF), para describir la sintaxis de un lenguaje. Permite representar gramáticas de libre contexto por medio de producciones, cada producción consta de reglas, símbolos terminales y no terminales los cuales son identificados previamente para el lenguaje de pseudocódigo.
- Visual Studio 2012 y Microsoft Net Framework, el cual provee las herramientas necesarias para implementar un compilador de pseudocódigo a partir de la gramática escrita en la notación BNF.
- DotNetBar WinForms el cual cuenta con más de 50 componentes impresionantes que ayudan a crear la interfaz de usuario profesional.
- DevExpressCodeRush, como herramienta para realizar pruebas unitarias y pruebas de sistemas.
- Computadora portátil con sus respectivos medios de almacenamiento y software necesario para el desarrollo del compilador.

4.6. Técnicas e instrumentos de recolección de datos

Las técnicas e instrumentos utilizados fueron:

- Registro de notas de la Asignatura de Algorítmica I.
- Encuesta de indicadores de usabilidad y ergonomía del compilador de pseudocódigo.

4.7. Procesamiento y análisis de datos

Para el procesamiento y análisis de datos se realizó:

- El procesamiento de los registros de notas de los dos grupos de la Asignatura de Algorítmica I (Semestres 2015 –I y 2015 - II); se realizó el cálculo en Excel.
- El análisis de los datos está basado en datos cuantitativos, para lo cual se usó la distribución t de Student, para dos poblaciones con muestras independientes.

CAPÍTULO V
RESULTADOS

5.1. Análisis e interpretación de los datos

5.1.1. Descripción de resultados de la hipótesis

a) Aprendizaje en el desarrollo de modularidad

A continuar se muestra los resultados de la cantidad de alumnos que han aprobado, desaprobado, reprobado y NSP en los semestres académicos 2015 – I y 2015 – II del curso de Algorítmica I. Cabe mencionar que la columna "Alumnos Semestre 2015 - I" de la Tabla Nro. 2 se obtuvo del Acta de notas del semestre académico 2015 – I, realizando el promedio de notas del Contenido conceptual – columna I3 y el Contenido procedimental - columna I3 (Anexo A).

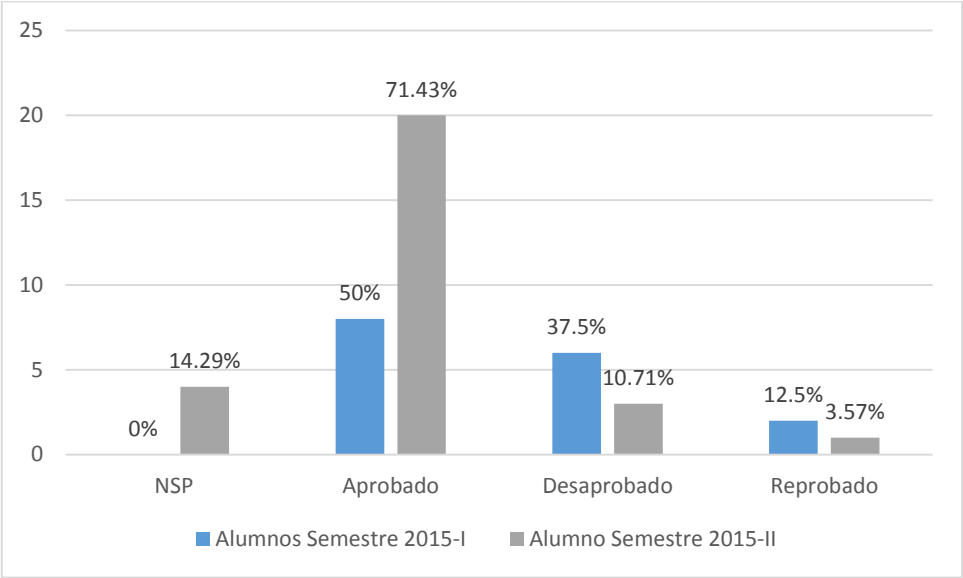
Del mismo modo la columna "Alumnos Semestre 2015 - II" de la Tabla Nro. 2 se obtuvo del Acta de notas del semestre académico 2015 – II, realizando el promedio de notas del Contenido conceptual – columna I3 y el Contenido procedimental - columna I3 (Anexo B).

Tabla N° 2: Aprendizaje en el desarrollo de modularidad de la asignatura de Algorítmica I

	Alumnos Semestre 2015-I	**%**	**Alumno Semestre 2015-II**	**%**
NSP	0	0 %	4	14.29 %
Aprobado	8	50 %	20	71.43 %
Desaprobado	6	37.5 %	3	10.71 %
Reprobado	2	12.5 %	1	3.57 %
Total	16	100 %	28	100 %

Fuente: Elaboración propia

Gráfico N° 1: Aprendizaje en el desarrollo de modularidad de la asignatura de Algorítmica I del semestre académico 2015-I y 2015-II



Fuente: Elaboración propia

Interpretación:

En la Grafico N° 1, se aprecia la cantidad de alumnos y los porcentajes de aprobados, desaprobados, reprobados y NSP de los semestre 2015 – I y 2015 - II. El porcentaje de aprobados del semestre académico 2015-I es de 50% los cuales no hicieron uso del compilador de pseudocódigo. Del semestre académico 2015-II, podemos apreciar que el porcentaje de alumnos aprobados es del 71.43% los cuales hicieron uso del compilador de pseudocódigo, lo cual determina que el uso del compilador de pseudocódigo mejoró el aprendizaje en el desarrollo de modularidad en la asignatura de Algorítmica I.

b) Aprendizaje en el desarrollo de procedimientos

A continuar se muestra los resultados de la cantidad de alumnos que han aprobado, desaprobado, reprobado y NSP en los semestres académicos 2015 – I y 2015 – II del curso de Algorítmica I. Cabe mencionar que la columna "Alumnos Semestre 2015 - I" de la Tabla Nro. 3 se obtuvo del Acta de notas del semestre académico 2015 – I,

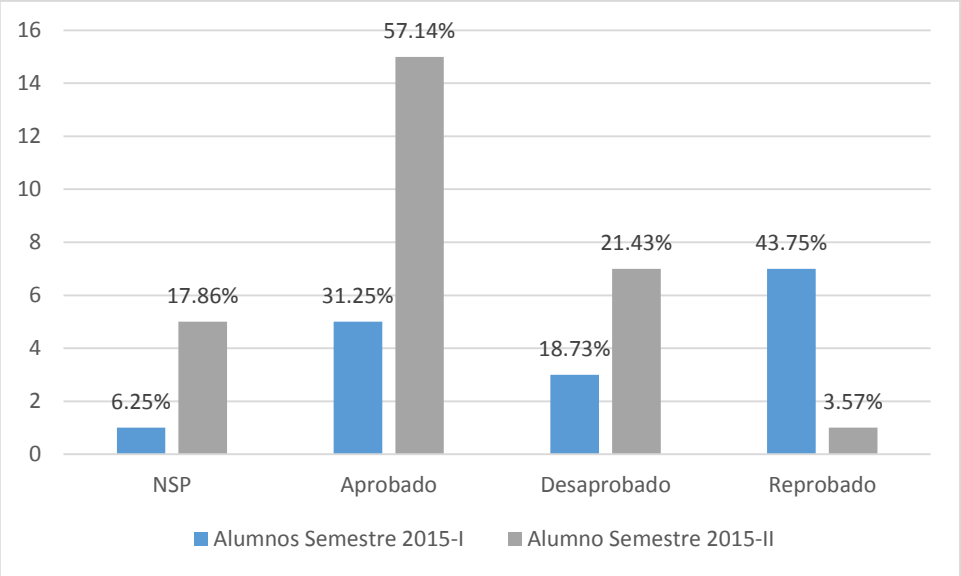
Contenido conceptual – columna I3 (Anexo A). Del mismo modo la columna "Alumnos Semestre 2015 - II" de la Tabla Nro. 3 se obtuvo del Acta de notas del semestre académico 2015 – II, Contenido conceptual - columna I3 (Anexo B).

Tabla N° 3: Aprendizaje en el desarrollo de procedimientos del curso de Algorítmica I

	Alumnos Semestre 2015-I	**%**	**Alumno Semestre 2015-II**	**%**
NSP	1	6.25 %	5	17.86 %
Aprobado	5	31.25 %	16	57.14 %
Desaprobado	3	18.73 %	6	21.43 %
Reprobado	7	43.75 %	1	3.57 %
Total	16	100%	28	100 %

Fuente: Elaboración propia

Gráfico N° 2: Aprendizaje en el desarrollo de procedimientos de la asignatura de Algorítmica I del semestre académico 2015-I y 2015-II



Fuente: Elaboración propia

Interpretación:

En la Grafico N° 2, se aprecia la cantidad de alumnos y los porcentajes de aprobados, desaprobados, reprobados y NSP de los semestre 2015 – I y 2015 - II. El porcentaje de aprobados del semestre académico 2015-I es de 31.25% los cuales no hicieron uso del compilador de pseudocódigo. Del semestre académico 2015-II, podemos apreciar que el porcentaje de alumnos aprobados es del 57.14% los cuales hicieron uso del compilador de pseudocódigo, lo cual determina que el uso del compilador de pseudocódigo mejoró el aprendizaje en el desarrollo de procedimientos en la asignatura de Algorítmica I.

c) Aprendizaje en el desarrollo de funciones

A continuar se muestra los resultados de la cantidad de alumnos que han aprobado, desaprobado, reprobado y NSP en los semestres académicos 2015 – I y 2015 – II del curso de Algorítmica I. Cabe mencionar que la columna "Alumnos Semestre 2015 - I" de la Tabla Nro. 4 se obtuvo del Acta de notas del semestre académico 2015 – I, Contenido procedimental – columna I3 (Anexo A).

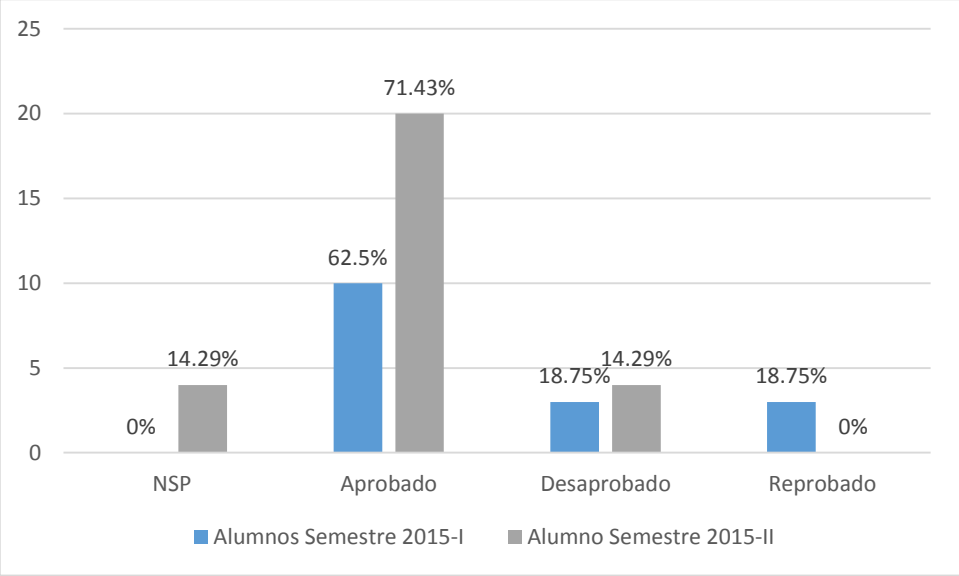
Del mismo modo la columna "Alumnos Semestre 2015 - II" de la Tabla Nro. 4 se obtuvo del Acta de notas del semestre académico 2015 – II, Contenido procedimental - columna I3 (Anexo B).

Tabla N° 4: Aprendizaje en el desarrollo de funciones de la asignatura de Algorítmica I

	Alumnos Semestre 2015-I	**%**	**Alumno Semestre 2015-II**	**%**
NSP	0	0 %	4	14.29 %
Aprobado	10	62.5 %	20	71.43 %
Desaprobado	3	18.75 %	4	14.29 %
Reprobado	3	18.75 %	0	0 %
Total	16	100 %	28	100 %

Fuente: Elaboración propia

Gráfico N° 3: Aprendizaje en el desarrollo de funciones de la asignatura de Algorítmica I del semestre académico 2015-I y 2015-II



Fuente: Elaboración propia

Interpretación:

En la Grafico N° 3, se aprecia la cantidad de alumnos y los porcentajes de aprobados, desaprobados, reprobados y NSP de los semestre 2015 – I y 2015 - II. El porcentaje de aprobados del semestre académico 2015-I es de 62.5% los cuales no hicieron uso del compilador de pseudocódigo. Del semestre académico 2015-II, podemos apreciar que el porcentaje de alumnos aprobados es del 71.43% los cuales hicieron uso del compilador de pseudocódigo, lo cual determina que el uso del compilador de pseudocódigo mejoro el aprendizaje en el desarrollo de funciones en la asignatura de Algorítmica I.

5.1.2. Contrastación de la hipótesis

a) Prueba de hipótesis general: Aprendizaje en el desarrollo de modularidad.

Planteamiento de la hipótesis estadística

H_0: $\mu_1 = \mu_2$ Las notas de la unidad de desarrollo de modularidad de la asignatura de Algorítmica I aplicando el compilador de pseudocódigo es igual a las notas de la unidad de desarrollo de modularidad de la asignatura de Algorítmica I sin aplicar el compilador de pseudocódigo.

H_1: $\mu_1 > \mu_2$ Las notas de la unidad de desarrollo de modularidad de la asignatura de Algorítmica I aplicando el compilador de pseudocódigo es mayor a las notas de la unidad de desarrollo de modularidad de la asignatura de Algorítmica I sin aplicar el compilador de pseudocódigo.

Nivel de significancia

En este caso de estudio se consideró un nivel de significancia de α=5%=0.05.

Estadístico de prueba

Para el análisis de la hipótesis general se aplicará la distribución t de Student para dos poblaciones con muestras independientes.

$$T_c = \frac{\mu_1 - \mu_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$

Dónde:

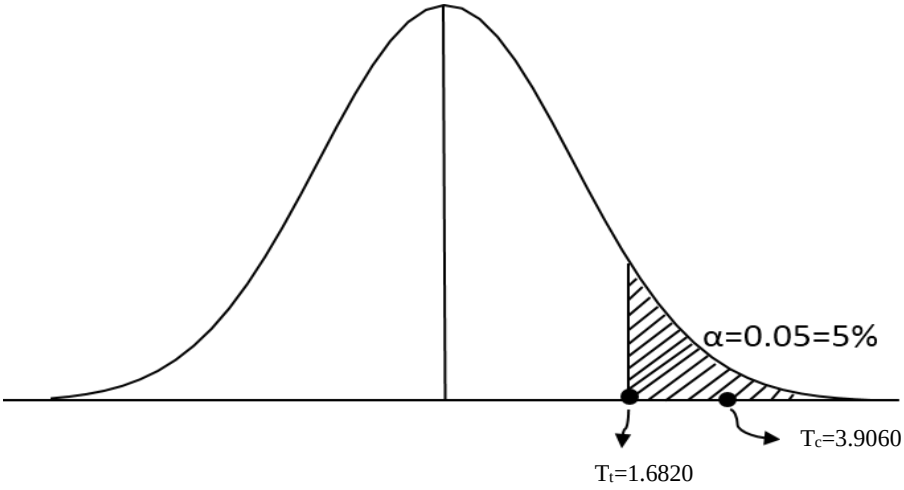
μ_1 y μ_2 : $Medias\ muestrales$

n_1 y n_2 : $Tamaño\ de\ muestras$

Reemplazando se obtienen los siguientes datos.

Análisis	Sin el Compilador	Con el Compilador
Media	10.59375	12.58333333
Varianza	11.07395833	8.427536232
Observaciones	16	28
Grados de libertad	42	
Estadístico t	1.6820	
Valor de Tc	3.9060	

Región crítica



Si $T_c > T_t$ entonces se rechaza la H_0 y aceptamos la H_1
Haciendo el cálculo de las muestra se obtiene Tc=3.9060. Al obtener un Tc > Tt se rechaza la hipótesis nula y se acepta la hipótesis alterna, por lo que podemos afirmar el compilador de pseudocódigo contribuye al aprendizaje en el desarrollo de modularidad en la asignatura de Algorítmica I.

b) Prueba de hipótesis específica: Aprendizaje en el desarrollo de procedimientos.

Planteamiento de la hipótesis estadística

H_0: $\mu_1 = \mu_2$ Las notas de la unidad de desarrollo de procedimientos de la asignatura de Algorítmica I aplicando el compilador de pseudocódigo es igual a las notas de la unidad de desarrollo de procedimientos de la asignatura de Algorítmica I sin aplicar el compilador de pseudocódigo.

H_1: $\mu_1 > \mu_2$ Las notas de la unidad de desarrollo de procedimientos de la asignatura de Algorítmica I, aplicando el compilador de pseudocódigo es mayor a las notas de la unidad de desarrollo de procedimientos de la asignatura de Algorítmica I sin aplicar el compilador de pseudocódigo.

Nivel de significancia

En este caso de estudio se consideró un nivel de significancia de α=5%=0.05

Estadístico de prueba

Para el análisis de la hipótesis general se aplicará la distribución t de Student para dos poblaciones con muestras independientes.

$$T_c = \frac{\mu_1 - \mu_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$

Dónde:

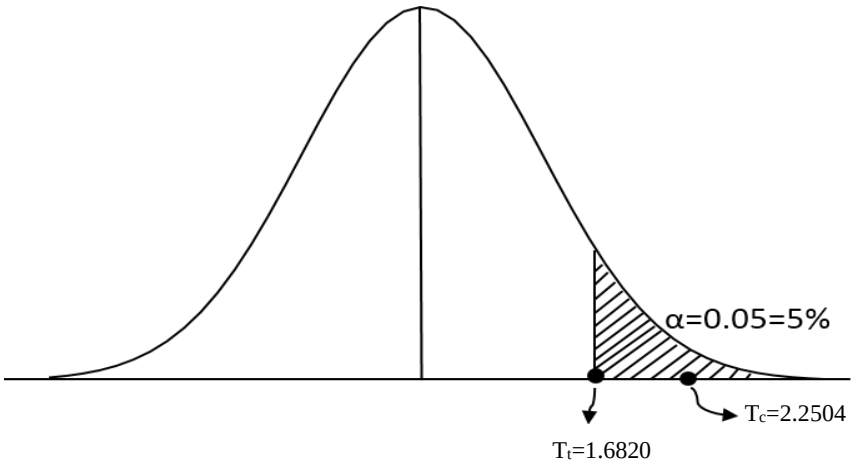
$\mu_1 \, y \mu_2 \, : Medias \; muestrales$

$n_1 \, y n_2 \, : Tamaño \; de \; muestras$

Reemplazando se obtienen los siguientes datos.

Análisis	Sin el Compilador	Con el Compilador
Media	8.6875	12.04166667
Varianza	25.42916667	15.17210145
Observaciones	16	28
Grados de libertad	42	
Estadístico t	1.6820	
Valor de Tc	2.2504	

Región crítica



Si Tc > Tt entonces se rechaza la H0 y aceptamos la H1

Haciendo el cálculo de las muestra se obtiene Tc=2.2504. Al obtener un Tc > Tt se rechaza la hipótesis nula y se acepta la hipótesis alterna, por lo que podemos afirmar el compilador de pseudocódigo contribuye al aprendizaje en el desarrollo de procedimiento en la asignatura de Algorítmica I.

c) Prueba de hipótesis específica: Aprendizaje en el desarrollo de funciones.

Planteamiento de la hipótesis estadística

H_0: $\mu_1 = \mu_2$ Las notas de la unidad de desarrollo de funciones de la asignatura de Algorítmica I aplicando el compilador de pseudocódigo es igual a las notas de la unidad de desarrollo de funciones de la asignatura de Algorítmica I sin aplicar el compilador de pseudocódigo.

H_1: $\mu_1 > \mu_2$ Las notas de la unidad de desarrollo de funciones de la asignatura de Algorítmica I aplicando el compilador de pseudocódigo es mayor a las notas de la unidad de desarrollo de funciones de la asignatura de Algorítmica I sin aplicar el compilador de pseudocódigo.

Nivel de significancia

En este caso de estudio se consideró un nivel de significancia de α=5%=0.05.

Estadístico de prueba

Para el análisis de la hipótesis general se aplicará la distribución t de Student para dos poblaciones con muestras independientes.

$$T_c = \frac{\mu_1 - \mu_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$

Dónde:

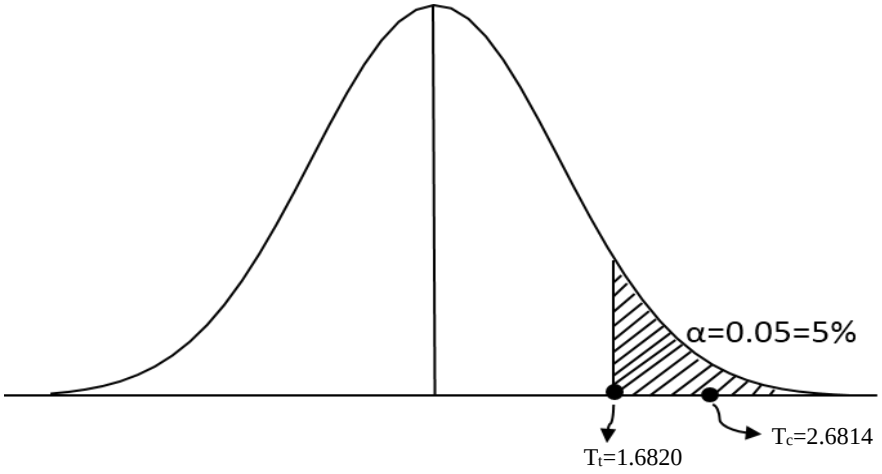
$\mu_1 \, y \, \mu_2 : Medias \; muestrales$

$n_1 \, y \, n_2 : Tamaño \; de \; muestras$

Reemplazando se obtienen los siguientes datos.

Análisis	Sin el Compilador	Con el Compilador
Media	12.5	13.125
Varianza	24.93333333	6.461956522
Observaciones	16	28
Grados de libertad	42	
Estadístico t	1.6820	
Valor de Tc	2.6814	

Región crítica



Si Tc > Tt entonces se rechaza la H0 y aceptamos la H1

Haciendo el cálculo de las muestra se obtiene Tc=2.6814. Al obtener un Tc > Tt se rechaza la hipótesis nula y se acepta la hipótesis alterna, por lo que podemos afirmar el compilador de pseudocódigo contribuye al aprendizaje en el desarrollo de funciones en la asignatura de Algorítmica I.

5.1.3. Resultados de la usabilidad

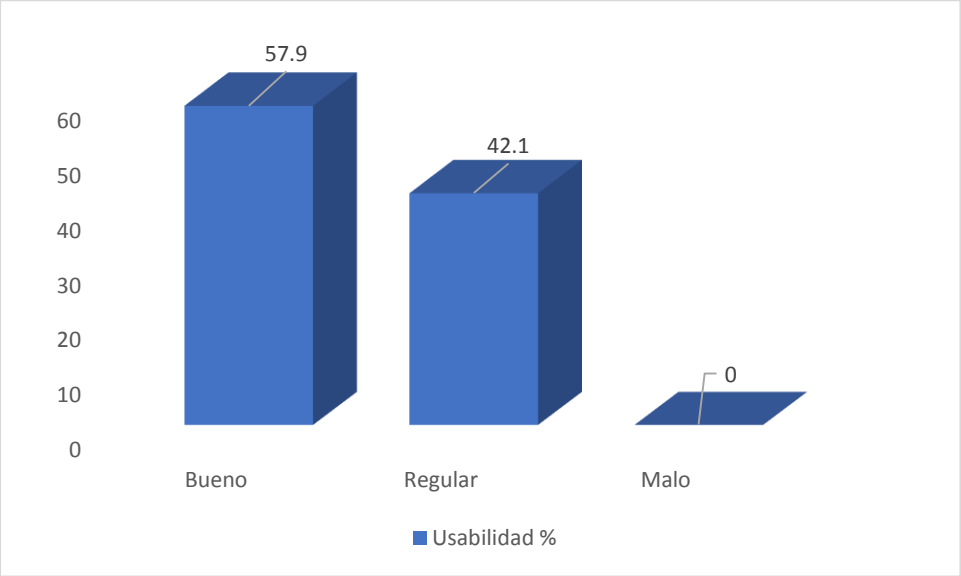
Los resultados de la ergonomía se obtuvieron en base a las respuestas de la encuesta realizada a los alumnos de la Asignatura de Algorítmica I (Anexo E). A continuación se muestra los resultados obtenido de la usabilidad del compilador de pseudocódigo.

Tabla N° 5: Resultado de usabilidad

	Usabilidad	
	n	%
Bueno	11	57.9
Regular	8	42.1
Malo	0	0
Total	19	100

Fuentes: Elaboración Propia

Gráfico N° 4: Resultado en porcentaje de usabilidad



Fuentes: Elaboración Propia

Interpretación: En el Gráfico N° 4, se aprecia los porcentajes de usabilidad con las escalas de bueno, regular y malo; Un 57.9% afirman que la usabilidad del compilador de pseudocódigo es bueno, un 42.1% afirma que fue regular y un 0% indica que fue malo.

5.1.4. Resultados de la ergonomía

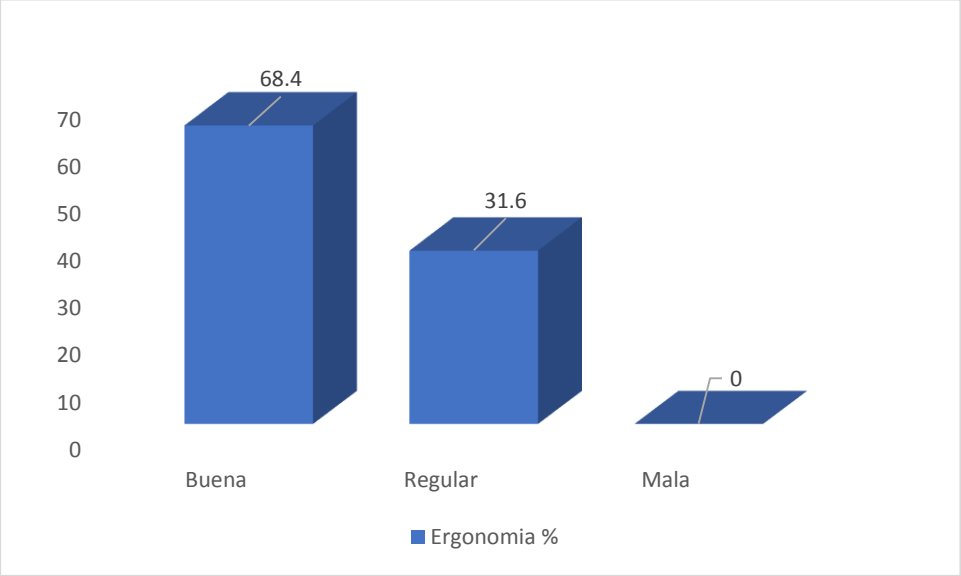
Los resultados de la ergonomía se obtuvieron en base a las respuestas de la encuesta realizada a los alumnos de la Asignatura de Algorítmica I (Anexo E). A continuación se muestra los resultados obtenido de la ergonomía del compilador de pseudocódigo.

Tabla N° 6 : Resultado de ergonomía.

	Ergonomía	
	n	%
Bueno	13	68.4
Regular	6	31.6
Malo	0	0
Total	19	100

Fuentes: Elaboración Propia

Gráfico N° 5 : Resultado en porcentaje de la ergonomía



Fuentes: Elaboración Propia

Interpretación: En el Gráfico N° 5, se aprecia los porcentajes de ergonomía con las escalas de bueno, regular y malo; Un 68.4% afirman que la ergonomía del compilador de pseudocódigo fue bueno, en tanto el 31.6% afirman que fue regular y un 0% afirma que fue malo.

5.2. Desarrollo del compilador de pseudocódigo en español

5.2.1. Planificación del proyecto

5.2.1.1. User Stories

Para la especificación de requerimientos se realizaron User Stories, tal como lo sugiere la metodología XP, sobre la cual está basada la metodología utilizada en este documento.

A continuación se enumeran los User Stories identificados:

1. Al cerrar la aplicación: Si el pseudocódigo en el que se está trabajando no se ha guardado, preguntar si se desean guardar los cambios. Se puede realizar desde alguna opción de la barra de menú.
2. Al abrir la aplicación: Aparece la ventana principal con la barra de menú, que permitirá la creación y edición de pseudocódigo.

3. Archivo: Se deben poder crear, abrir y editar archivos de pseudocódigo.
4. Definición de archivos de Pseudocódigo: Archivos propios del compilador con extensión (.hito) y archivos planos de texto, que contienen un código fuente comprensible para la herramienta, la cual estará en la capacidad de realizar sobre estos análisis léxico, sintáctico y semántico, y ofrecerá la funcionalidad de compilación.
5. Archivos .hito: Los archivos con extensión hito (.hito); permitirán abrir automáticamente con el compilador y tendrán su propio icono.
6. Barra de menú de comandos y operadores y funciones: Permite agregar fácilmente códigos predefinidos para una rápita edición.
7. Vista de creación archivo de Pseudocódigo: Cuando se crea un archivo de pseudocódigo, la aplicación debe mostrar un editor del código, y debe proveer la estructura básica de un código fuente, realizando la coloración del texto.
8. Vista de edición de archivo de Pseudocódigo: Para editar un archivo de Pseudocódigo se debe mostrar el editor de código, junto con un grupo de herramientas, definidas a libre disposición del desarrollo, que promuevan la usabilidad de la herramienta.
9. Editar fuente de letra: La fuente se puede editar con las opciones que se encuentran en una barra ubicada en la parte superior de la herramienta en la cual se puede subir o bajar el tamaño de la fuente, cambiar el tipo de fuente.
10. Guardar archivos: Para guardar los archivos se puede realizar por medio del icono en la barra de herramientas o desde la opción de la barra de menú, se puede guardar los cambios en el mismo documento o realizar una copia del mismo.
11. Imprimir archivo: Se debe proveer la funcionalidad de impresión de archivos de código fuente así como de vista previa de impresión.
12. Verificar Sintaxis: Se puede realizar desde la barra de herramientas y desde la opción de la barra menú, haciendo click en el icono. En esta se realiza la estructura del programa a ejecutar y se muestran posibles errores sintácticos y semánticos que posea el código fuente escrito.

13. Ejecutar: Se puede realizar desde la barra de herramientas y desde la opción de la barra menú. En esta se realizará primero la compilación del archivo fuente, para después ser ejecutado el programa en caso que no haya errores.
14. Vista de errores: Debe mostrar un cuadro en el cual serán mostrados los errores para que se puedan corregir y así ejecutar el programa.
15. Ayuda: Se puede acceder a la ayuda desde la opción de la barra de menú y desde el icono de la barra de herramientas, en esta se encontrará lo necesario para utilizar la herramienta adecuadamente.
16. Estilos: Permite al editor cambiar el aspecto de la interfaz a diferentes estilos.
17. Traducción del Pseudocódigo: Se debe tener la opción de guardar el pseudocódigo a C#, C++ y Java.
18. Ejemplos: Se puede acceder a los ejemplos desde la opción del menú Archivo, en esta se encontrará los ejemplos aplicativos para utilizar en la herramienta del compilador.

5.2.1.2. Aspectos técnicos generales y requerimientos no funcionales

Una vez definidos los requerimientos funcionales de la herramienta a través de los User Stories, se determinaron los aspectos generales de implementación, y que a su vez mostraron los requerimientos no funcionales de la misma.

A. Ambiente de ejecución

Para ofrecer una suficiente flexibilidad a la herramienta, se sugirió inicialmente que ésta pudiese ser ejecutada en el sistema operativo Windows. Para esto, se especificó que la herramienta se ejecutara en los sistemas operativos instalado en los laboratorios de cómputo, siendo estos Windows XP y versiones superiores.

Así, entonces se determinó que la herramienta esté disponible en un asistente de instalador que permitiera realizar la instalación completa del compilador; se escogió como lenguaje de programación C#, ya que permite tener un ambiente de desarrollo completo.

B. Requerimientos Técnicos

Debido a la implementación de la herramienta, es necesario que las máquinas donde se instalaron el compilador de pseudocódigo tengan instalado el .NET Framework 2.0 o versiones superiores. Así, la herramienta se podrá ejecutar en el sistema operativo Windows XP o sus versiones superiores.

5.2.1.3. Roles XP

Tabla N° 7: Roles XP

Jefe del Proyecto
Juan Carlos Muñoz Miranda
Cliente
Alumnos de la asignatura de Algorítmica I
Equipo de trabajo
Juan Carlos Muñoz Miranda
Coach
Ing. Ecler Mamani Vilca

Fuente: Elaboración propia

5.2.1.4. Definiciones y acrónimos

Definiciones

- **Interfaces:** Medio que permite la comunicación entre el usuario y el sistema.
- **Net Framework:** Es un framework de Microsoft que permite un rápido desarrollo de aplicaciones.
- **ArrayList:** Es una clase en C# que permite almacenar datos en memoria de forma similar a los Array, con la ventaja de que el número de elementos es de forma dinámica.
- **HITO:** Nombre del compilador del pseudocódigo
- **DotNetBar:** Conjunto de librerías con las que cuenta Visual Studio para el diseño de la GUI
- **JFLEX:** Generador de analizador léxico
- **CUP:** Generador de analizador sintáctico

Acrónimos:

- **GUI:** Interfaz gráfica de usuario
- **BNF:** Extened Backus Naus Form
- **DFAs:** Autómatas finitos determinísticos
- **DS:** Diagrama de sintaxis

5.2.1.5. Herramientas tecnológicas utilizadas

Tabla N° 8: Herramientas tecnológicas utilizadas

Elemento	Herramienta Utilizada
IDE para el desarrollo	Visual Studio 2012
Diseño de interfaz	DotNetBar WinForms
Gráficos estadísticos	Microsoft Excel 2013
Control de versiones	Tortoise SVN
Lenguaje de programación	CSharp
Documentación	Microsoft word y excel 2013
Estimación del proyecto	Microsoft Project

Fuente: Elaboración propia

5.2.2. Diseño

Una vez definidos los requerimientos técnicos y funcionales de la herramienta, se procedió a realizar el diseño de ésta, a través múltiples iteraciones que permitieran el trazado de los requerimientos con sus respectivos componentes de software.

5.2.2.1. Modelo de la arquitectura

Para el desarrollo del compilador se utilizó el MVC (Modelo-Vista-Controlador), ya que su uso es común en aplicaciones interactivas, orientada a su uso por parte de usuarios finales, y que se encuentra dividida en tres grandes componentes.

El componente del Modelo contiene la funcionalidad principal y la manutención lógica de los datos. El componente de la Vista muestra la información al usuario en una forma que éste pueda comprender, y el componente Controlador se ocupa de procesar la entrada del usuario, para modificar o procesar el Modelo, y a su vez actualizar los datos que la Vista presenta.

5.2.2.2. Diagrama de Clases

El diagrama de clases permite determinar todas las clases que intervienen en el desarrollo de un flujo de trabajo del compilador y como se encuentran relacionadas entre sí.

Figura N° 2: Lista de clases del compilador de pseudocódigo

Fuente: Elaboración propia

Diagrama de Componentes

A continuación se muestra el diagrama de componentes (Figura Nro. 3) utilizados en el compilador:

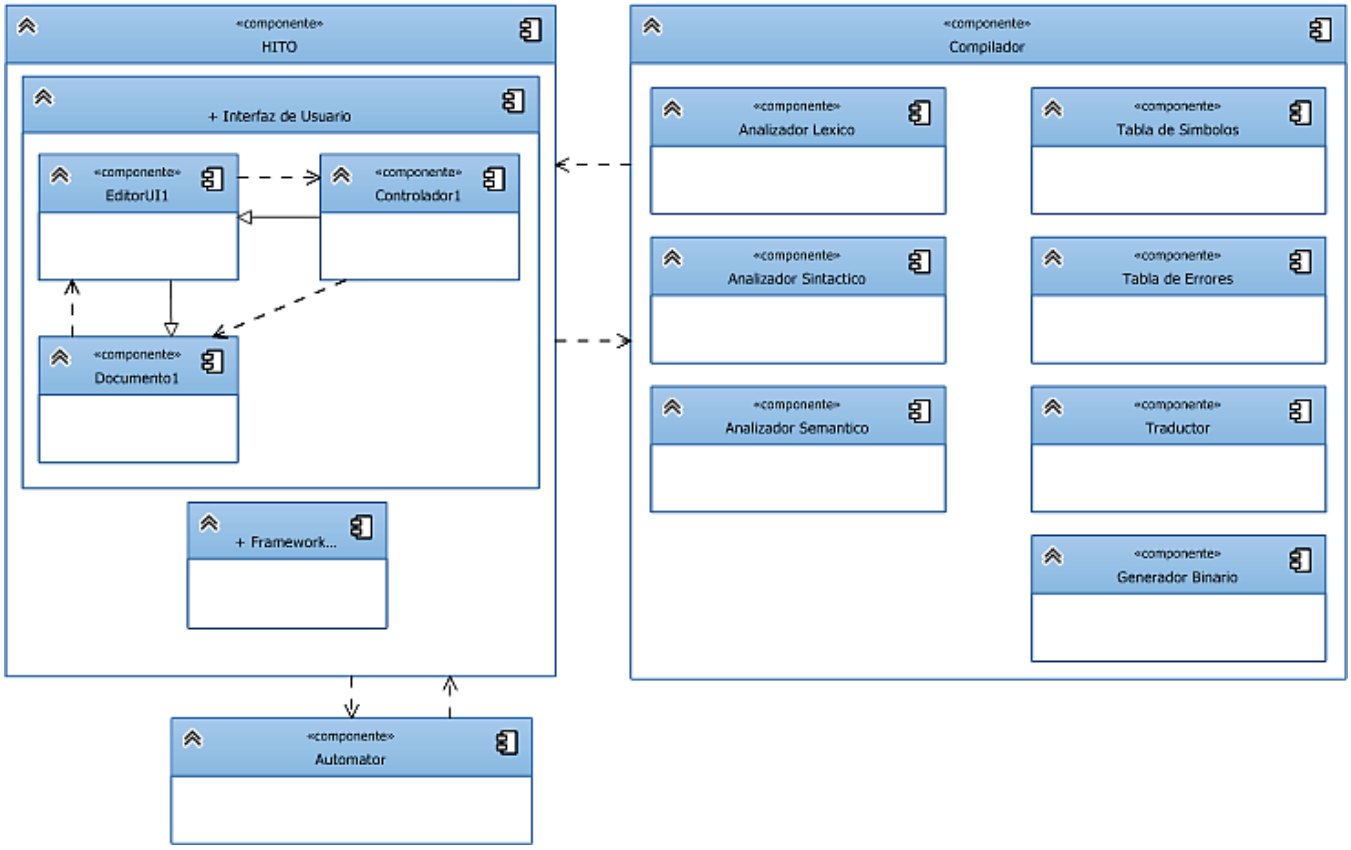


Figura N° 3: Diagrama de Componentes

Fuente: Elaboración propia

Según el diagrama proporcionado, los componentes de la herramienta se describen así:

- Componente HITO: Es el componente base de la herramienta. Se encarga de administrar todos los componentes en la aplicación, y se encuentra constituido por la interfaz gráfica y el Net Framework.
- Componente Interfaz de Usuario: Se encarga de administrar y proveer la interacción con el usuario. Se conforma de tres subcomponentes que actúan como el modelo arquitectónico.
- Componente Documento: Es la representación lógica de los datos (Modelo) que el usuario crea y edita. Los documentos serán los Archivos de Pseudocódigo.

- ❖ Componente EditorUI: Se encarga de proveer la visualización de los datos hacia el usuario (Vista), dándole una representación gráfica de los documentos que se hayan creado o se estén editando. Pueden existir varias perspectivas para un mismo documento.
- ❖ Componente Controlador: Administra el intercambio de vistas y la actualización de los documentos según los cambios realizados por el usuario (Controlador).
- ➢ Framework: Componente destinado a hacer la herramienta flexible y extensible, con la finalidad de cargar dinámicamente las librerías usadas por la herramienta.
- ➢ Componente Compilador: Se encarga de la manipulación lógica de los documentos de Pseudocódigo y de realizar la fase de análisis competente del proceso de compilación, con el fin de generar el árbol sintáctico, realizar la generación de código y la fase de síntesis del proceso de compilación, realizando la traducción del árbol sintáctico hacia el lenguaje de alto nivel definido.

  Es importante resaltar el componente compilador no fueron creados usando una herramienta de generación de compiladores tal como JFLEX, CUP, YACC o JavaCC.

5.2.2.3. Diseño Detallado y Codificación de la Aplicación

El desarrollo del compilador fue llevado a cabo utilizando como herramientas de programación el entorno de desarrollo integrado Visual Studio 2012. Se utilizó también un estándar común de codificación. El código fue administrado por un Servidor de control de versiones SubVersion.

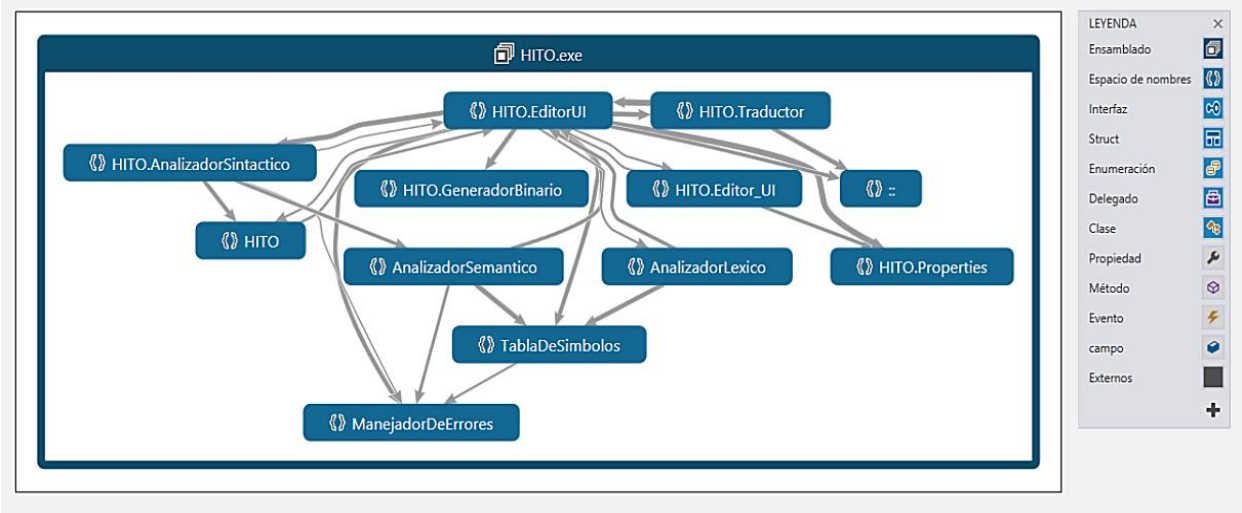


Figura N° 4: Diagrama de dependencias Editor HITO

Fuente: Elaboración propia

A. Componente EditorUI

El componente de EditorUI fue desarrollado utilizando patrones de diseño generales para aplicaciones de escritorio, de la siguiente forma:

- Se utilizaron Clases para definir cada uno de los puntos de acceso general a los componentes de la aplicación, tal como lo muestra la figura 6.

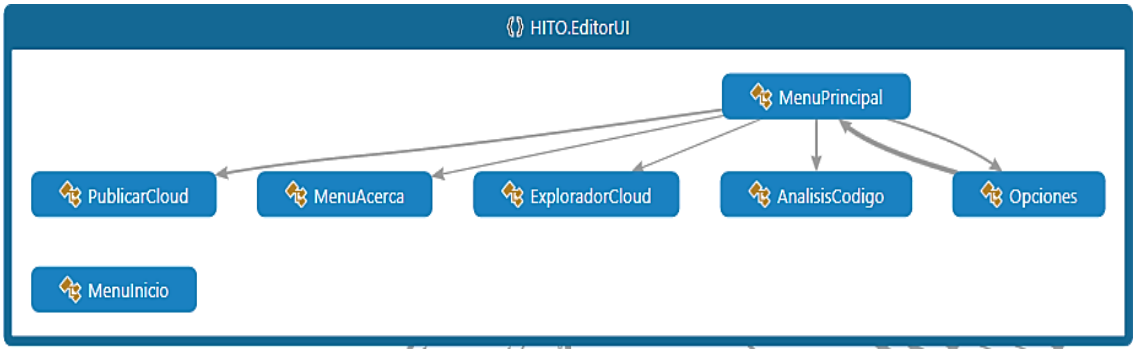


Figura N° 5: Diagrama de dependencias del EditorUI

Fuente: Elaboración propia

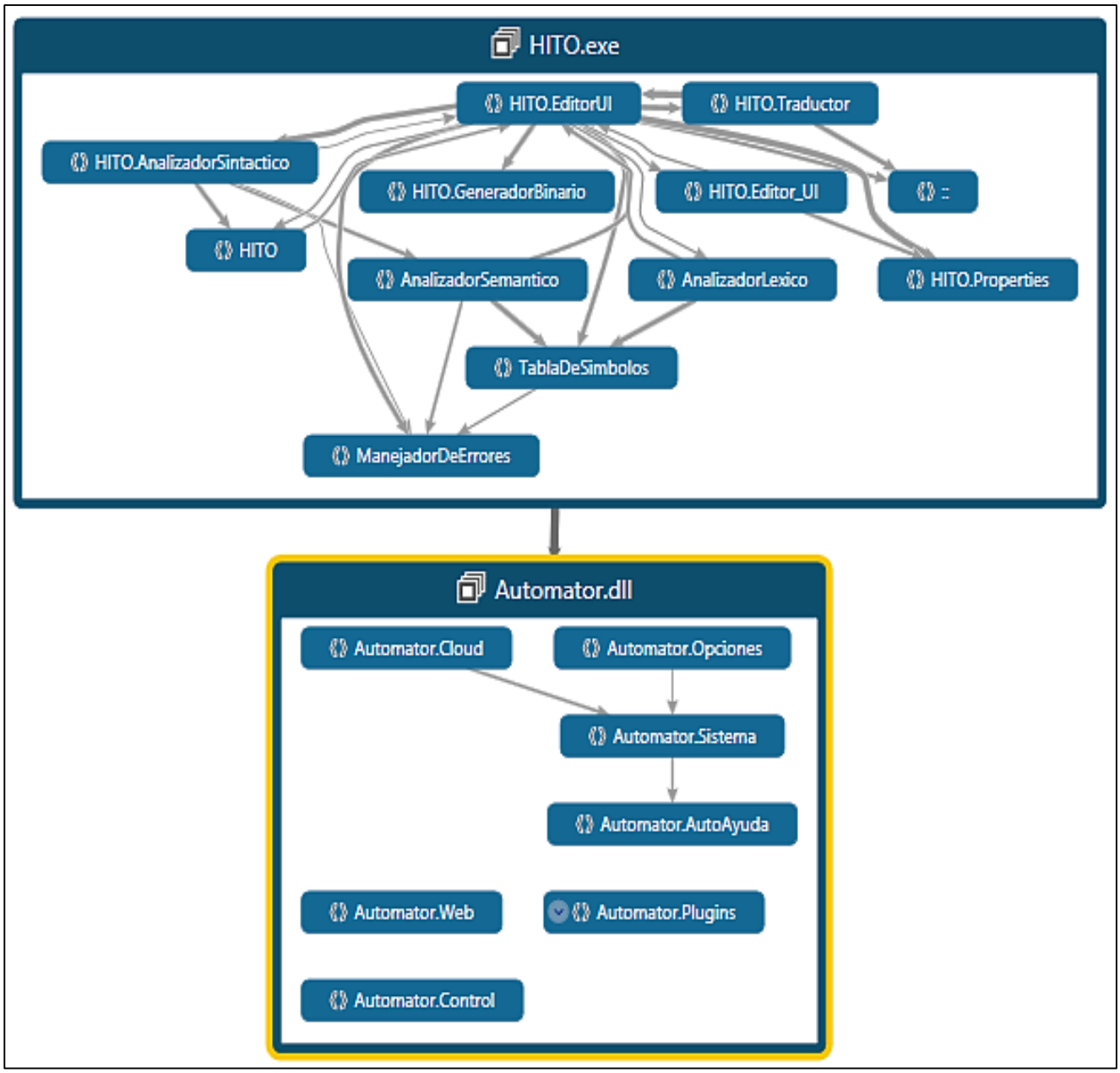


Figura N° 6: Diagrama de dependencias del EditorUI y el Automator

Fuente: Elaboración propia

Se utilizó los diferentes componentes, para la implementación de las funcionalidades de la barra de menú del EditorUI, tales como compilar, verificar sintaxis, comandos, operadores, herramientas, ayuda, etc.

B. Componente Automator

El componente Automator fue diseñado y codificado con la finalidad de ser reusable como una librería dll. Las clases implementada en el automator se encuentra en la Figura Nro.7.

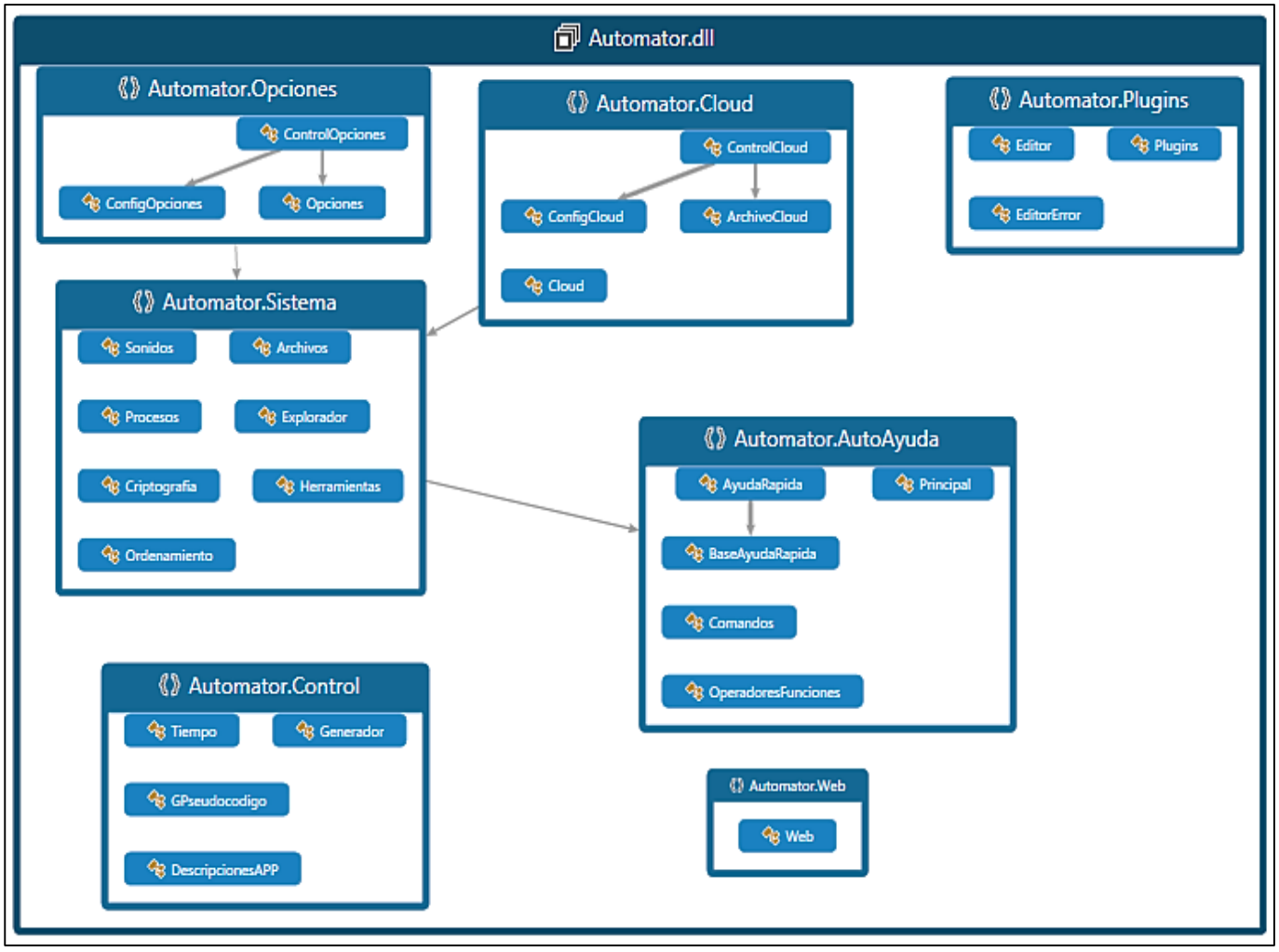


Figura N° 7: Diagrama de dependencias de la librería dll del automator

Fuente: Elaboración propia

C. Componente Analizador Léxico

Este componente se implementó siguiendo los lineamientos propuestos en el proceso de traducción descrito en el marco conceptual. El análisis léxico es el proceso que sirve para identificar que los símbolos de entrada pertenecen al alfabeto de un lenguaje dado; lo anterior aplicado al entorno del compilador, se refiere a que ésta etapa debe analizar que el programa a compilar contenga caracteres del lenguaje español, así como números y símbolos permitidos por el lenguaje definido.

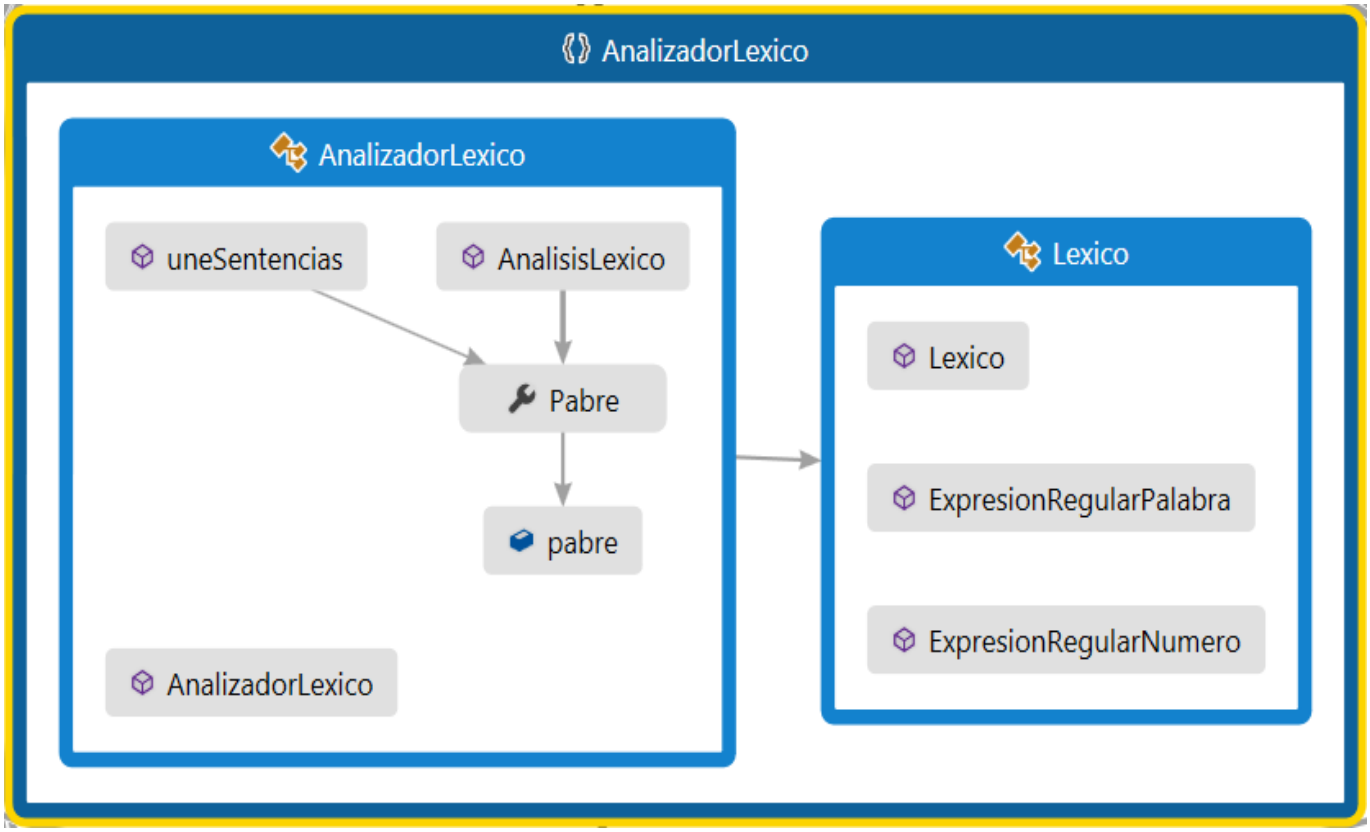


Figura N° 8: Diagrama de dependencias del Analizador Léxico
Fuente: Elaboración propia

D. Componente Analizador Sintáctico

La segunda etapa fue el analizador Sintáctico del compilador, el cual permite identificar que las palabras del lenguaje (tokens) siguen correctamente una regla de producción, al momento que el analizador léxico devuelve tokens.

Para implementar un Analizador Sintáctico, se utilizan gramáticas libres de contexto, las cuales se denotan por medio de producciones; cual es el orden que debe seguir cierta secuencia de tokens, para lo anterior se utilizaron los árboles de derivación como un método de ayuda para verificar que una gramática acepta la secuencia de tokens.

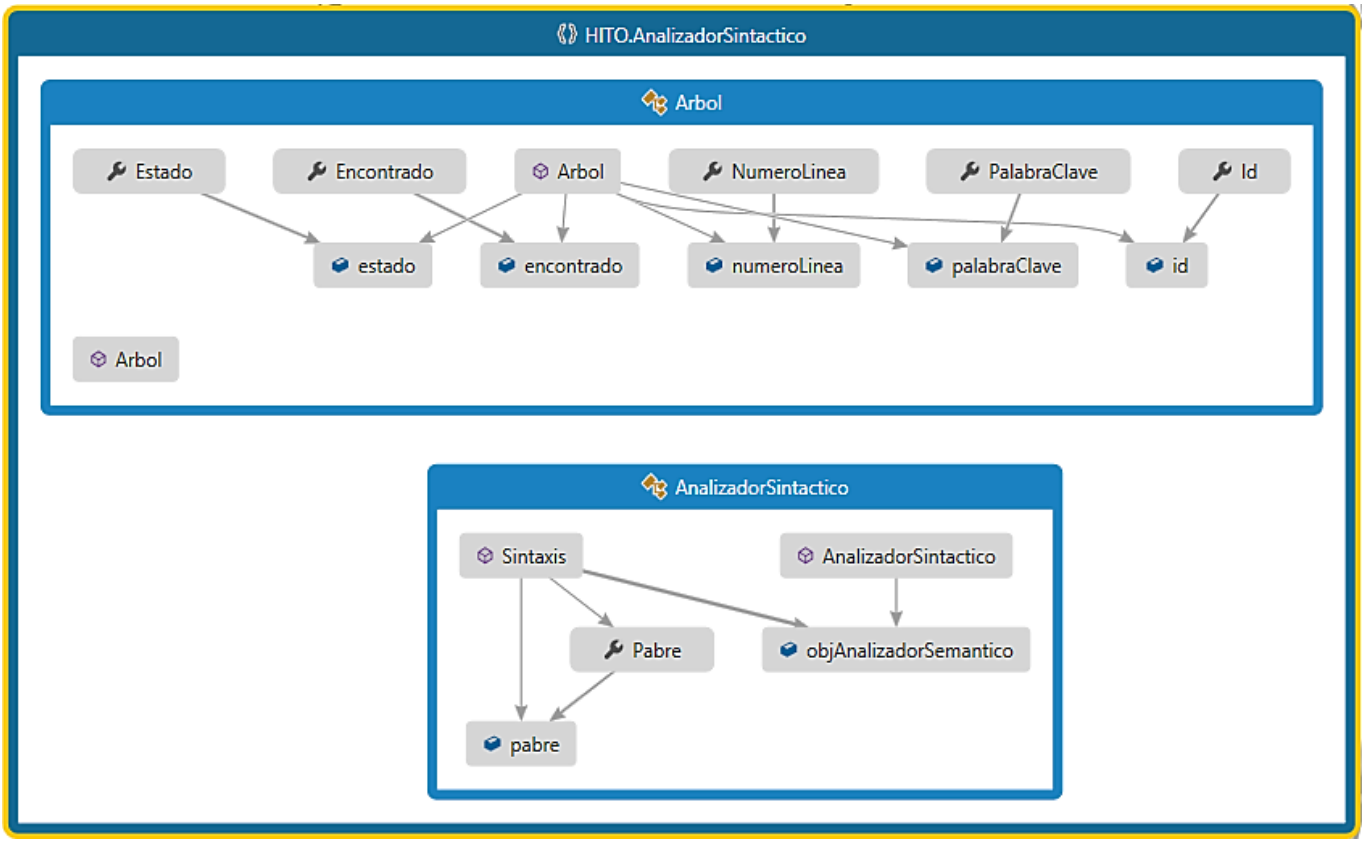


Figura N° 9: Diagrama de dependencias del Analizador Sintáctico

Fuente: Elaboración propia

E. Componente Analizador Semántico

El análisis semántico se encarga de validar aspectos que el análisis sintáctico es incapaz de verificar, por ejemplo la comprobación de tipos y comprobación del alcance o "scope" de las variables utilizadas a lo largo del programa.

El Analizador Semántico del pseudocódigo, recorren el árbol sintáctico generado durante la etapa de análisis sintáctico. El analizador semántico trabaja directamente con las expresiones regulares para verificar los tipos y alcances de las variables declaradas en el pseudocódigo como se muestra en la figura 10.

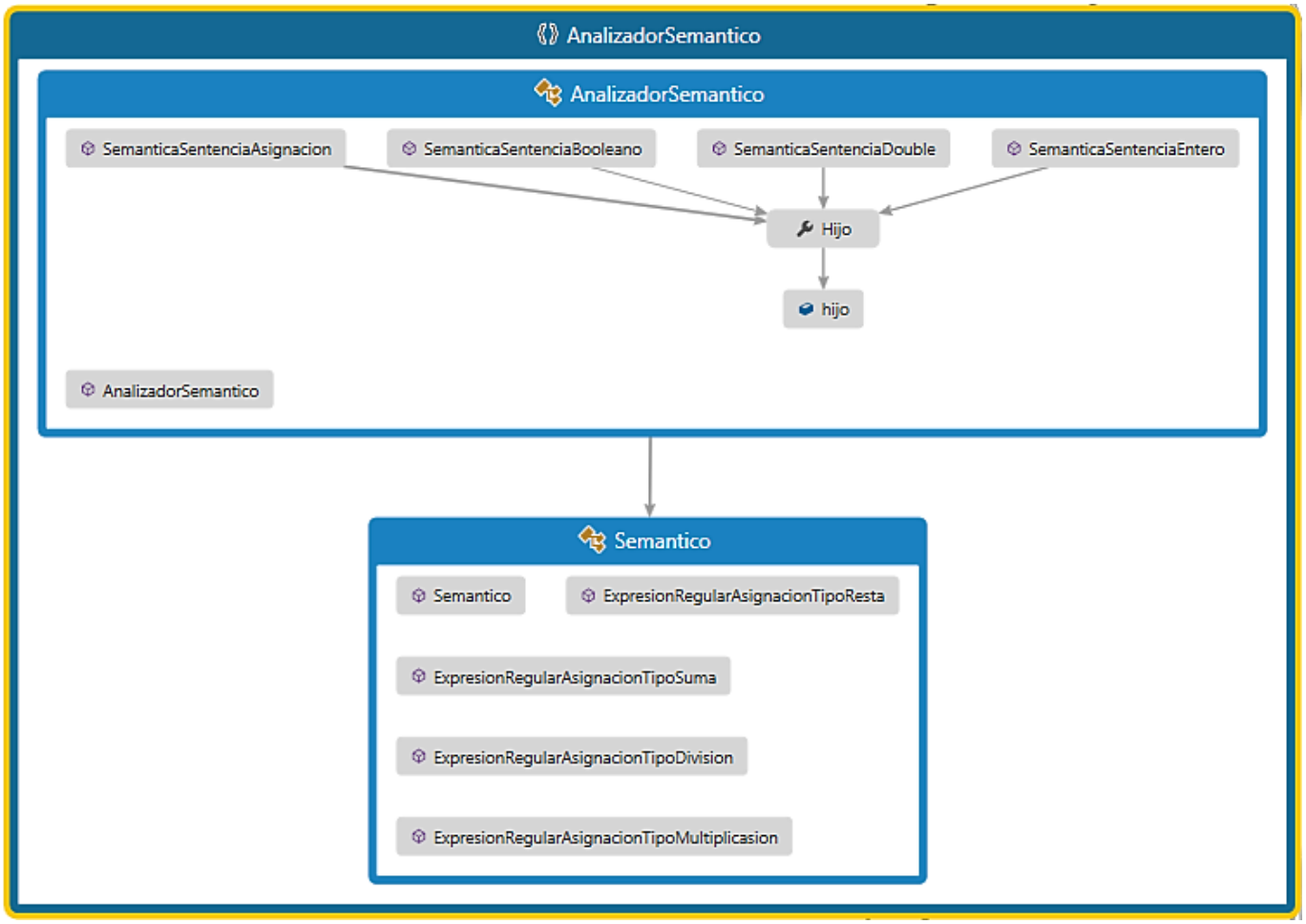


Figura N° 10: Diagrama de dependencias del Analizador Semántico

Fuente: Elaboración propia

F. Componente Tabla de Símbolos

La tabla de símbolos es el componente con el que trabaja el Analizador Léxico, Sintáctico y Semántico y su tarea principal es generar las entradas de las variables y subrutinas declaradas en el programa dentro del compilador; además verifica si el identificador de una variable ya ha sido declarado previamente en el programa, lo anterior con la finalidad de evitar declaraciones múltiples del mismo identificador y para asegurar que no se usan identificadores que no han sido declarados.

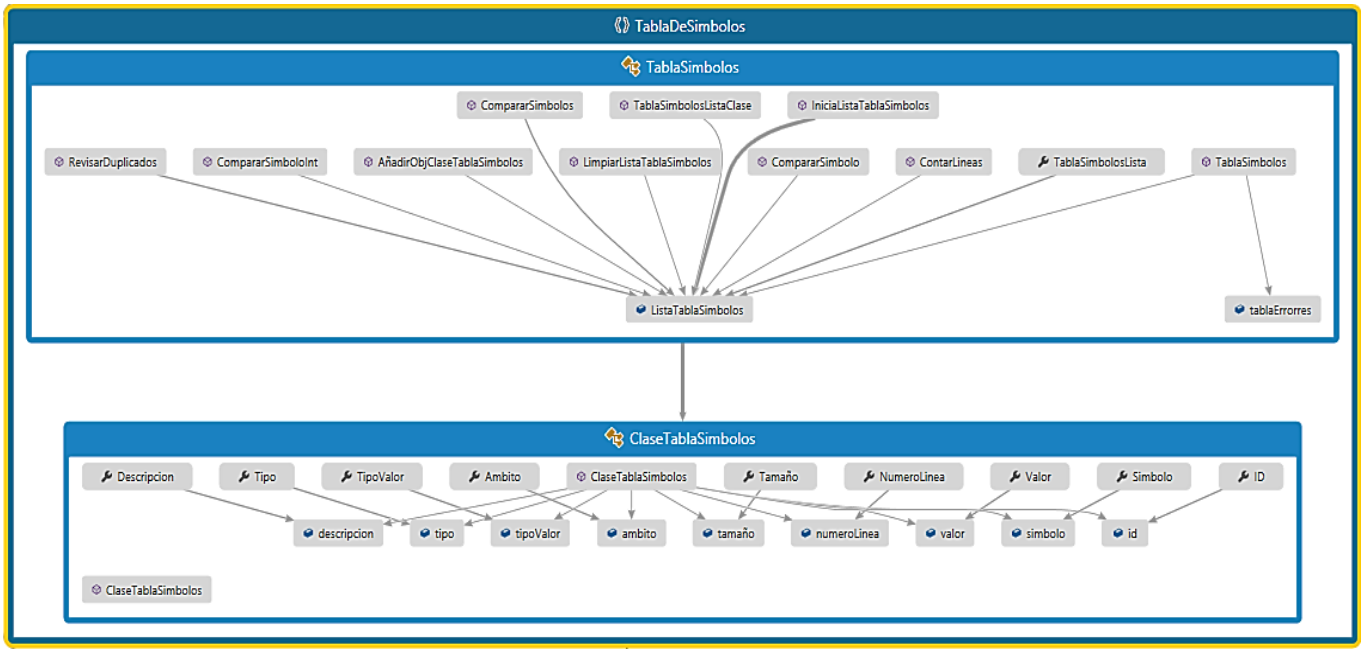


Figura N° 11: Diagrama de dependencias de la Tabla de Símbolos

Fuente: Elaboración propia

Cada una de las entradas de la tabla de símbolos contiene la siguiente información:

- ID o identificador de la variable o subrutina
- El tipo de dato de la variable o subrutina; el tipo de la subrutina es el tipo de dato que
- Descripción de la variable o subrutina
- En la tabla de símbolos toma un tipo de dato "void" (solamente se utiliza éste tipo de dato dentro de la tabla de símbolos, no es un tipo que se pueda usar dentro del programa en la declaración de variables).
- El valor con el que fue inicializada la variable. Cuando se agrega una subrutina, ésta propiedad toma un valor nulo.
- Tipo de valor con el que fue inicializada la variable.
- El ámbito de la variable declarada.
- Símbolo utilizado en la inicialización de la variable
- El tamaño de la variable inicializada.
- El número de línea donde se encuentre en el código fuente.

G. Componente Manejador de errores

El Manejador de errores se encarga de verificar cada uno de los errores encontrados en cada una de las fases de compilación del pseudocódigo.

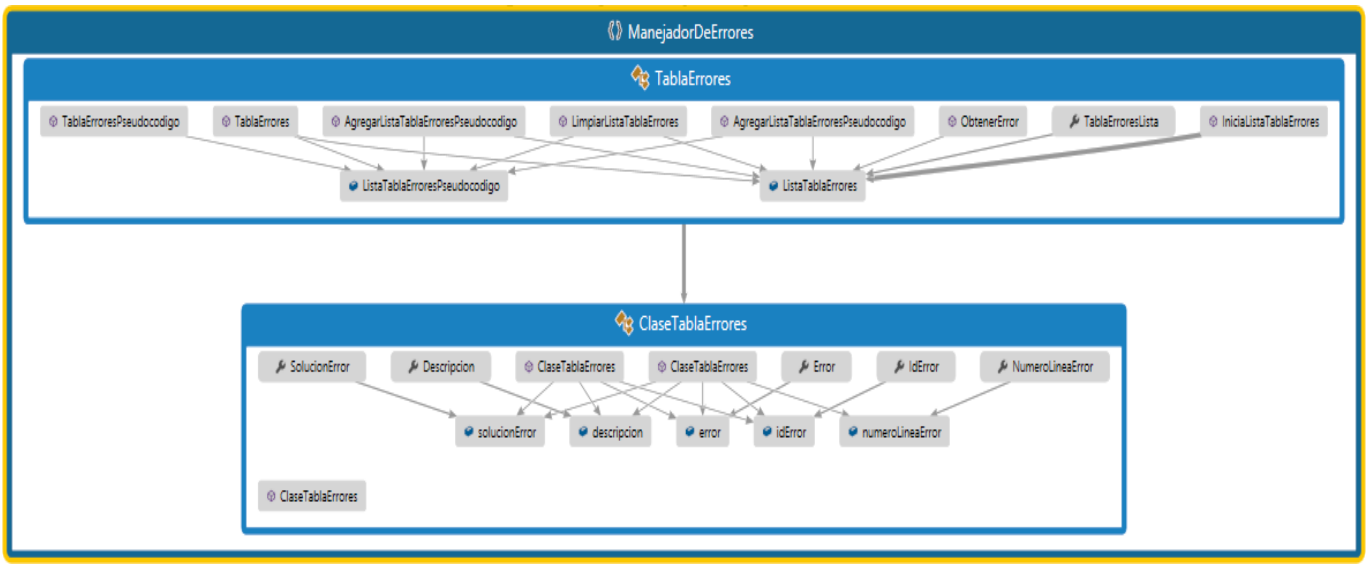


Figura N° 12: Diagrama de dependencias del Manejador de Errores

Fuente: Elaboración propia

H. Componente Generador de Código

El generador de código traduce el código fuente del Pseudocódigo en español al código destino, en éste caso a código C#. Al igual que los componentes del Analizador Semántico, el Generador de Código realiza un recorrido en el árbol sintáctico para traducir cada uno de los nodos. El código C# se genera al finalizar el analizador semántico, se guarda en un archivo con extensión .cs el cual está listo para ser compilado usando la librería System.CodeDom.Compiler.

I. Componente Compilador

El componente compilador fue desarrollado con el fin de ofrecer una capa de abstracción entre el árbol sintáctico y la generación del código destino. La idea primordial del funcionamiento del compilador reside en la implementación de un traductor. Así, se logra ocultar la implementación de la traducción al lenguaje de alto nivel y su compilación y ejecución por parte de este componente, ofreciendo una mayor flexibilidad a la herramienta.

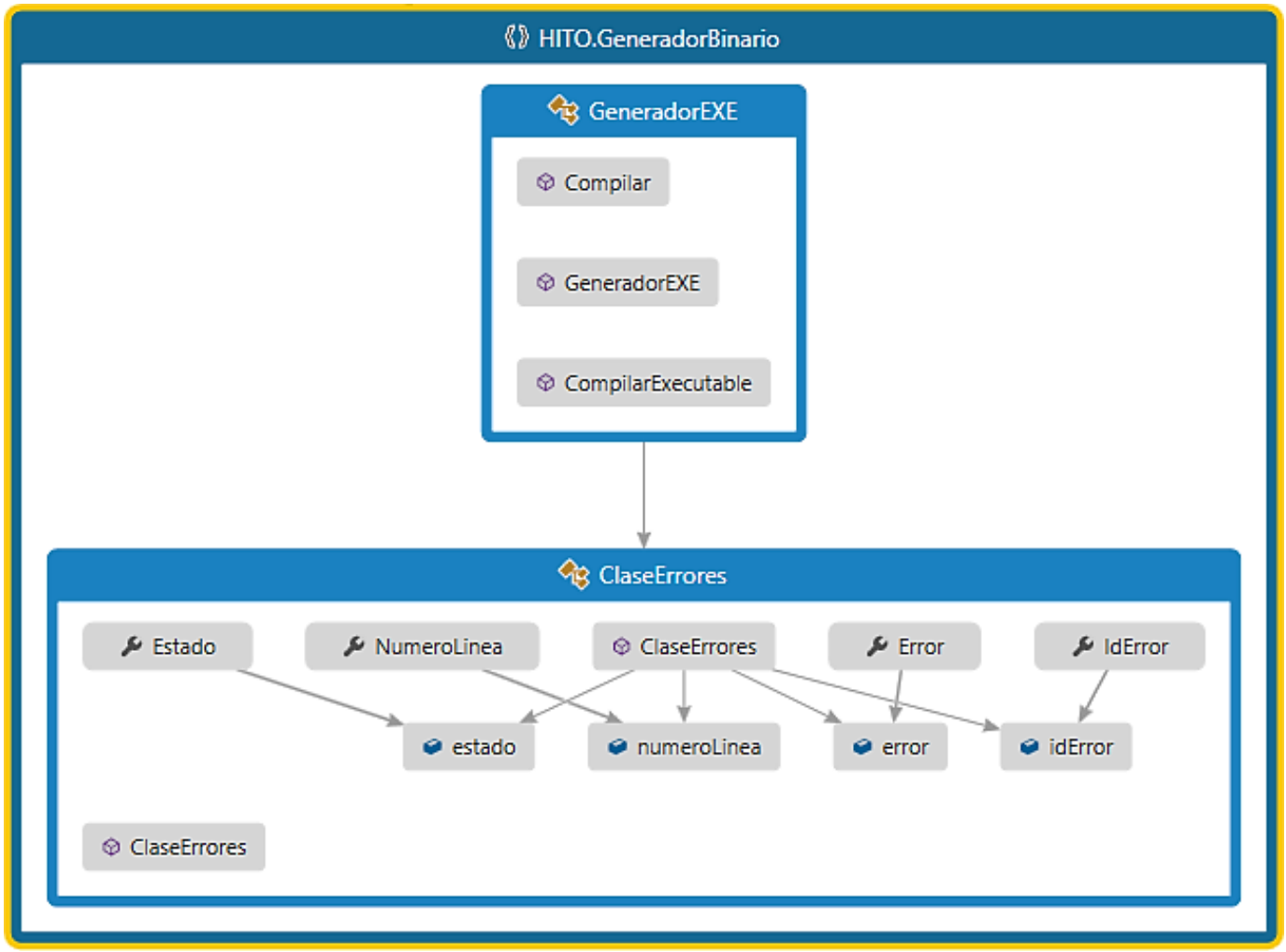


Figura N° 13: Diagrama de dependencias del Compilador

Fuente: Elaboración propia

5.2.2.4. Proceso de traducción y compilación del pseudocódigo

Después de ser definidos los componentes de la herramienta, se estableció que el proceso de traducción y compilación del pseudocódigo fuese el siguiente:

1. El pseudocódigo en texto es pasado al compilador como una cadena de caracteres.
    a. Al recibir la cadena de caracteres, el compilador pasa sobre ésta el analizador léxico, el cual busca e identifica los distintos lexemas definidos en la tabla de símbolos, utilizando para esto un motor de expresiones regulares.
    b. Una vez identificados los lexemas, son almacenados y revisados por un Parser, que actúa también como analizador sintáctico. Éste se encarga de la construcción del árbol sintáctico, identificando los posibles errores que existiesen en la sintaxis del pseudocódigo escrito.
    c. Con el árbol sintáctico construido, el compilador pasa sobre éste el analizador semántico, el cual se encarga de buscar y reportar errores en el significado de las sentencias escritas.
2. Una vez procesado el texto, y transformado en el árbol sintáctico, éste es pasado hacia el compilador, para generar los objetos ejecutables
    a. El Compilador utiliza un traductor quien se encarga de transformar el árbol sintáctico en código de algún lenguaje de alto nivel (la implementación actual utiliza C#), y almacenarlo dentro de un archivo de C# (extensión de archivo .cs).
    b. Con el archivo fuente generado, éste es pasado hacia el compilador del lenguaje de alto nivel (la implementación actual utilizo para éste propósito Net Framework), para generar el archivo ejecutable.
    c. Ya generado los ejecutables, se enlaza y ejecuta estos archivos para ser manipulados por el usuario.

5.2.2.5. Diseño del compilador

A. Lenguaje del Compilador

El Lenguaje para la ejecución de algoritmos en pseudocódigo se basa específicamente en las características empleadas para el pseudocódigo en el lenguaje castellano, donde han sido aplicadas las adaptaciones necesarias que brindan al lenguaje la rigurosidad suficiente para ser considerado un lenguaje de programación, principalmente respecto a una definición estricta de su lexicografía y sintaxis, y a un establecimiento claro de su semántica.

B. Lenguaje Hito

Se denominará Lenguaje Hito al lenguaje resultante luego de aplicar todas las modificaciones que sean determinadas como necesarias durante el proceso de revisión del Lenguaje para la ejecución de algoritmos en pseudo-código.

C. Modificaciones del Lenguaje Hito con respecto a PSeInt

A continuación se listan todas las modificaciones en la estructura y sintaxis del lenguaje para la ejecución de algoritmos en pseudocódigo la cual no provee de las funcionalidades en el PSeInt.

1. Eliminación de la declaración previa de sub-algoritmos

Al momento de especificar la definición de los sub-algoritmos, se utilizará la firma del sub-algoritmo para realizar conjuntamente su declaración y registro en la tabla de símbolos.

Estado anterior: antes de comenzar la definición de sub-algoritmos (incluyendo el sub-algoritmo principal) se debe declarar todos los sub-algoritmos para posteriormente establecer su definición luego de haber definido al sub-algoritmo principal.

Justificación:

- ✓ Es un mecanismo más intuitivo para principiantes de la programación.
- ✓ Es un mecanismo que cuenta con poca dificultad de implementación.

2. Adición de las palabras clave "función" y "procedimiento" para la definición de sub-algoritmos.

Al momento de definir un sub-algoritmo éste podrá especificarse de dos maneras:

Funcion <tipo> <Nombre Función> (<parámetros>)
<instrucciones>
retornar <valor>
FinFuncion

O bien:

Procedimiento <Nombre Función> (<parámetros>)
<instrucciones>
FinProcedimiento

Estado anterior: la sintaxis para la definición de un sub-algoritmo era una sola para funciones y procedimientos:

[<tipo>] <Nombre Función> (<parámetros>)
<instrucciones>
[retornar <valor>]
FinSubAlgoritmo

Justificación:

Es mucho más claro e intuitivo para un estudiante de programación que los sub-algoritmos se vean diferenciados de esta manera. Además, facilita el aprendizaje en cuanto a las diferencias conceptuales y utilitarias entre las funciones y los procedimientos.

3. Adición de envío de parámetros por valor o por referencia.

Al enviar un parámetro dentro de una llamada a un sub-algoritmo se podrá colocar de manera opcional la nueva palabra reservada referencia de la siguiente manera:

<nombreSubalgoritmo>([referencia] <parametro>,….)

El colocar esta palabra reservada indicará que el parámetro se envía por referencia mientras que su ausencia indica que el parámetro se envía por valor.

Estado anterior: no existía ningún mecanismo dentro del lenguaje para la ejecución de algoritmos en pseudo-código que permitiera establecer esta diferenciación. Por defecto todos los parámetros se enviaban por valor.

Justificación: Diferenciar entre envío de parámetros por referencia y envío de parámetros por valor es un elemento importante para un estudiante de programación y se considera esencial asimilar este recurso desde el comienzo del aprendizaje.

4. Adición de funciones matemáticas y de manejo de cadenas y caracteres.

Las funciones agregadas son las siguientes:

- Funciones matemáticas que permitan obtener: valor de π, potencias, logaritmos, redondeo, valor absoluto, raíz cuadrada, senos, cosenos, tangentes, exponencial, etc.
- Funciones de cadena y carácter para concatenar, subcadena longitud, mayúsculas, minúsculas, remover, remplazar, comparar, etc.
- Funciones que permita realizar graficas desde el editor de pseudocódigo como: dibujar punto, dibujar línea, dibujar imagen, dibujar texto, dibujar rectángulo, dibujar arco, etc.
- Una función de tiempo que permite contar el tiempo en milisegundos.

Justificación: Poder realizar estas operaciones amplía en gran medida el tipo de programas que pueden ser escritos con el lenguaje para la ejecución de algoritmos en pseudo-código, lo cual se traduce en una ventaja para los estudiantes de programación, ya que expande sus posibilidades creativas y mejora sus procesos de aprendizaje.

5.2.2.6. Definición de la gramática libre de contexto

La gramática libre de contexto que genera al lenguaje HITO se presenta a continuación descrita en notación BNF (Backus Naus Form):

Algoritmo =

"Inicio" Bloque "Fin" Declaracion_Variable_global Subalgoritmo

Declaracion_Variable_global =

"global" Tipo identificador ["[" entero ["," entero] "]"] {"," identificador ["[" entero ["," entero]"]"]}.

Declaracion_Variable =

Tipo identificador ["[" entero ["," entero] "]"] {"," identificador ["[" entero ["," entero]"]"]}.

Tipo =

"entero"|"real"|"cadena"|"caracter"

Subalgoritmo =

Firma_Subalgoritmo Bloque

Firma_Subalgoritmo =

("Procedimiento" | "Funcion" Tipo) identificador "(" [Parametro {"," Parametro}] ")".

Parametro =

Tipo identificador ["[" "#" ["," "#"] "]"].

Bloque =

Declaracion_Variable Instrucción

Instrucción =

Llamada_Subalgoritmo | Asignacion | Instruccion_Si| Instruccion_Segun | Instruccion_Mientras|Instruccion_Repetir | Instruccion_Para | Instruccion_E/S | retornar" Expresion

Instrucción_E/S =

Instruccion_Leer | Instruccion_Escribir | Intruccion_Pausa

Variable =

Identificador ["[" Expresion_Numerica ["," Expresion_Numerica] "]"]

Llamada_Subalgoritmo =

identificador "(" [(["referencia"] Variable | Expresion) {"," (["referencia"] Variable | Expresion)}] ")".

Asignacion =

Variable "=" Expresion

Instruccion_Si =

"Si" "(" Expresion_Numerica ")" "Entonces" Instruccion ["sino" Instruccion] "FinSi".

Instruccion_Segun =

"Segun" " (" Variable")" "Hacer"{"caso" Expresion_Numerica ":" Instruccion";"["romper;"]} "defecto"":" Instruccion";"["romper;"]

Instruccion_Mientras =

"Mientras" "(" Expresion_Numerica ") Entonces" Instruccion "FinMientras".

Instruccion_Repetir =

"Repetir" Instruccion "Hasta Que" "(" Expresion_Numerica ");"

Instruccion_Para =

"Para(" Variable "=" Expresion_Numerica ";" Expresion_Relacional ";"

Expresion_Numerica ")Hacer" Instruccion "FinPara".

Instruccion_Leer =

"Leer" "(" Variable ");"

Instruccion_Escribir =

"Escribir" "(" Expresion ");"

Instruccion_Pausa =

"Pausa" "(" ");"

Expresion =

Dato_Caracter | Dato_Cadena | Expresion_Numerica.

Expresion_Numerica =

Expresion_Conjuncion {"|" Expresion_Conjuncion}.

Expresion_Conjuncion =

Expresion_Relacional {"&" Expresion_Relacional}.

Expresion_Relacional =

["!"] Expresion_Aritmetica ["<"|">"|"<="|">="|"=="|"!=" Expresion_Aritmetica].

Expresion_Numerica =

Termino {"+"|"-" Termino}.

Termino =

Factor {"*"|"/" Factor}.

Factor =

["+"|"-"] (entero | real | Variable | Llamada_Subalgoritmo | Funcion_Numerica|

Funcion_Numerica_Cadena_Caracter | "(" Expresion_Numerica ")").

Funcion_Numerica =

("Tangente" |"Seno" |"Coseno" |"Arctan"| "Arcsen"| "Arccos"|

"Redordeo"| "ABS" | "SQRT" | "redondear"| log"| "ln" | "exp"

"(" Expresion_Numerica ")") | ("pi" "(" ")") | "finalArchivo" |

(("mod" | "potencia" | "div") "(" Expresion_Numerica "," Expresion_Numerica ")")

Funcion_Cadena =

("concatenar" "(" Dato_Cadena "," Dato_Cadena ")") | ("extraerCadena"

"(" Dato_Cadena "," Expresion_Numerica "," Expresion_Numerica ")")|

("cadenaMayusculas" "(" Dato_Cadena ")")| ("cadenaMinusculas"

"(" Dato_Cadena ")")

Funcion_Numerica_Cadena_Caracter =

("compararCadenas" "(" Dato_Cadena "," Dato_Cadena ")")| ("buscarCadena"

"(" Dato_Cadena "," Dato_Cadena ")")| ("longitudCadena" "(" Dato_Cadena ")") |

("compararCaracteres" "(" Dato_Caracter "," Dato_Caracter ")").

Dato_Caracter =

caracter | Variable | Llamada_Subalgoritmo.

Dato_Cadena =

cadena | Variable | Llamada_Subalgoritmo | Funcion_Cadena

5.2.3. Codificación

5.2.3.1. Analizador Léxico

A. Lectura del archivo fuente

Se ha determinado que para la lectura del archivo fuente que contiene el programa escrito en el lenguaje HITO se utilizará una técnica de buffering con un buffer de entrada. La estrategia de esta técnica contará con los siguientes elementos:

- Una función encargada de la extracción de caracteres línea por línea desde el archivo fuente que contiene el programa escrito en el lenguaje HITO para su almacenamiento en el buffer de entrada.
- Esta función será invocada cada vez que se determine que el buffer de entrada de datos ha sido vaciado.
- Una función encargada de la extracción uno a uno de los caracteres almacenados en el buffer de entrada.
- Esta función será invocada continuamente durante el proceso de análisis lexicográfico del archivo fuente, de forma que, por medio de los caracteres extraídos se vayan formando los tokens que se utilizarán en el análisis sintáctico.

B. Expresiones regulares

a. Identificadores

identificador = letra(letra+digito+simbolo)*

Donde,

letra = {a, b,…, ñ, …, z, A, B, …,Ñ, ..., Z, á, é, í, ó, ú} y

digito={0,1,2,…,9}

simbolo={ _ } (guión bajo)

b. Números enteros

entero = digito(digito)*

digito={0,1,2,3,4,5,6,7,8,9}

c. Números reales

real = entero.entero

d. Cadena

cadena = “(letra+digito+simbolo)*”

Donde,

letra = {a, b,…, ñ,…, z, A, B, …,Ñ, ..., Z, á, é, í, ó, ú}

digito={0,1,2,…,9}

simbolo={x / x es cualquier carácter imprimible distinto de letra, dígito o “}

e. Carácter

caracter = ‘(letra+digito+simbolo)’

Donde,

letra = {a, b,…, ñ,…, z, A, B, …,Ñ, ..., Z, á, é, í, ó, ú}

digito={0,1,2,…,9}

simbolo={x / x es cualquier carácter imprimible distinto de letra, dígito o ‘}

C. Utilización de tokens

Uno de los objetivos principales del compilador, en la lectura del archivo fuente, es poder identificar dentro de él todos los tokens definidos para el lenguaje HITO. Los tokens de un lenguaje son precisamente los significados que se le otorgan a los lexemas que lo conforman. Dichos lexemas son secuencias de caracteres en el código fuente que definen las palabras reservadas, operadores y símbolos que tienen un significado especial para cierto lenguaje de programación.

Las palabras reservadas del lenguaje HITO pueden verse en la Tabla Nro. 16 del Anexo G y la definición de sus respectivos lexemas en el código fuente del compilador se colocarán dentro de un ArrayList, este se denominará ListaPalabrasReservadas y tendrá un tamaño que aumenta dinámicamente cuando es necesario.

La definición de dicho arreglo se muestra a continuación:

```
ArrayList ListaPalabrasReservadas = new ArrayList();
ListaPalabrasReservadas.Add("entero");
ListaPalabrasReservadas.Add("real");
ListaPalabrasReservadas.Add("cadena");
ListaPalabrasReservadas.Add("caracter");
ListaPalabrasReservadas.Add("vacio");
ListaPalabrasReservadas.Add("booleano");
ListaPalabrasReservadas.Add("constante");
ListaPalabrasReservadas.Add("nuevo");
ListaPalabrasReservadas.Add("referencia");
ListaPalabrasReservadas.Add("verdadero");
```

```
ListaPalabrasReservadas.Add("falso");
ListaPalabrasReservadas.Add("Inicio");
ListaPalabrasReservadas.Add("Fin");
ListaPalabrasReservadas.Add("Escribir");
ListaPalabrasReservadas.Add("Leer");
ListaPalabrasReservadas.Add("Si");
ListaPalabrasReservadas.Add("Entonces");
ListaPalabrasReservadas.Add("Sino");
ListaPalabrasReservadas.Add("FinSi");
ListaPalabrasReservadas.Add("Segun");
ListaPalabrasReservadas.Add("Hacer");
ListaPalabrasReservadas.Add("caso");
ListaPalabrasReservadas.Add("defecto");
ListaPalabrasReservadas.Add("romper");
ListaPalabrasReservadas.Add("continue");
ListaPalabrasReservadas.Add("FinSegun");
ListaPalabrasReservadas.Add("Mientras");
ListaPalabrasReservadas.Add("FinMientras");
ListaPalabrasReservadas.Add("Repetir");
ListaPalabrasReservadas.Add("Para");
ListaPalabrasReservadas.Add("FinPara");
ListaPalabrasReservadas.Add("Funcion");
ListaPalabrasReservadas.Add("retornar");
ListaPalabrasReservadas.Add("FinFuncion");
ListaPalabrasReservadas.Add("Procedimiento");
ListaPalabrasReservadas.Add("FinProcedimiento");
ListaPalabrasReservadas.Add("ABS");
ListaPalabrasReservadas.Add("Redondeo");
ListaPalabrasReservadas.Add("SQRT");
ListaPalabrasReservadas.Add("Seno");
ListaPalabrasReservadas.Add("Coseno");
ListaPalabrasReservadas.Add("Tangente");
ListaPalabrasReservadas.Add("ASeno");
ListaPalabrasReservadas.Add("ACoseno");
ListaPalabrasReservadas.Add("ATangente");
ListaPalabrasReservadas.Add("Logaritmo10");
ListaPalabrasReservadas.Add("Logaritmo");
ListaPalabrasReservadas.Add("Exponencial");
ListaPalabrasReservadas.Add("Potencia");
ListaPalabrasReservadas.Add("Aleatorio");
ListaPalabrasReservadas.Add("ConvertirEntero");
ListaPalabrasReservadas.Add("ConvertirReal");
ListaPalabrasReservadas.Add("ConvertirCadena");
ListaPalabrasReservadas.Add("ConvertirCaracter");
ListaPalabrasReservadas.Add("Mayuscula");
ListaPalabrasReservadas.Add("Minuscula");
ListaPalabrasReservadas.Add("Longitud");
ListaPalabrasReservadas.Add("SubCadena");
ListaPalabrasReservadas.Add("Concatenar");
ListaPalabrasReservadas.Add("Insertar");
ListaPalabrasReservadas.Add("Remover");
ListaPalabrasReservadas.Add("Remplazar");
ListaPalabrasReservadas.Add("Comparar");
ListaPalabrasReservadas.Add("global");
ListaPalabrasReservadas.Add("Tiempo");
ListaPalabrasReservadas.Add("DibujarPunto");
ListaPalabrasReservadas.Add("DibujarLinea");
ListaPalabrasReservadas.Add("DibujarLinea");
ListaPalabrasReservadas.Add("DibujarTexto");
ListaPalabrasReservadas.Add("DibujarRectangulo");
ListaPalabrasReservadas.Add("DibujarArco");
ListaPalabrasReservadas.Add("PI");
```

```
ListaPalabrasReservadas.Add("E");
ListaPalabrasReservadas.Add("Pause");
```

Una vez definidas los lexemas de las palabras reservadas, se utiliza una función para poder analizar el código fuente; para que el compilador pueda diferenciar las palabras reservadas con respecto a los símbolos y operadores especiales del lenguaje, se utiliza una función que inicializa con los tokens correspondientes haciendo uso del ArrayList definido anteriormente y la tabla de símbolos y el manejador de errores.

A continuación se presenta la definición de la función:

```
public string[] SentenciasLexicos(int CantidadLineas){
        string sentencia = null;
        string[] sentencias = new string[CantidadLineas];
        int bandera = 0;
        string tipov = "";
        ArrayList ListaP= PalabrasResevadas();
        for (int i = 1; i < CantidadLineas; i++){
           foreach (var token in ((MenuPrincipal)Pabre).
           tabla_simbolos.TablaSimbolosListaClase()){
                if (token.NumeroLinea == i && token != null){
                    for (int l = 1; l < ListaP.Count; l++){
                        string reservada =ListaP[l].ToString();
                        if (bandera == 0 && Regex.IsMatch(token
                         .Simbolo, @""+reservada+"$")){
                            token.TipoValor = token.Simbolo;
                            tipov = token.Simbolo;
                        }
                    }
                    if (bandera != 0){
                         sentencia=sentencia + " "
                         +token.Simbolo.ToString();
                        token.TipoValor = tipov;
                    }else{
                         sentencia = sentencia +
                         token.Simbolo.ToString();
                        bandera = 1;
                    }
                }
            }
            sentencias[i] = sentencia;
            sentencia = null;
            bandera = 0;
            tipov = "";
        }
        return sentencias;
    }
```

D. Obtener los identificadores, números, cadenas y caracteres

La estrategia de obtención de tokens, que se implementó en el scanner, se basa en autómatas finitos determinísticos (DFAs-por sus siglas en inglés), dado que son utilizados para reconocer expresiones regulares.

- DFA utilizado para reconocer un identificador o una palabra reservada en el scanner

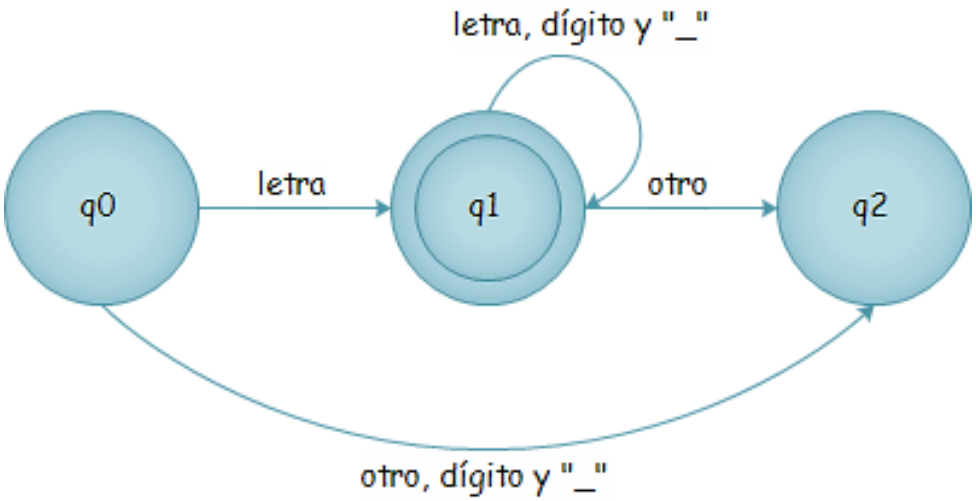


Figura N° 14: DFA para identificadores y palabras reservadas

Fuente: Elaboración propia

q0: Estado inicial.

q1: Estado de aceptación de un identificador.

q2: Estado de rechazo.

- DFA utilizado para reconocer un número entero o un número real.

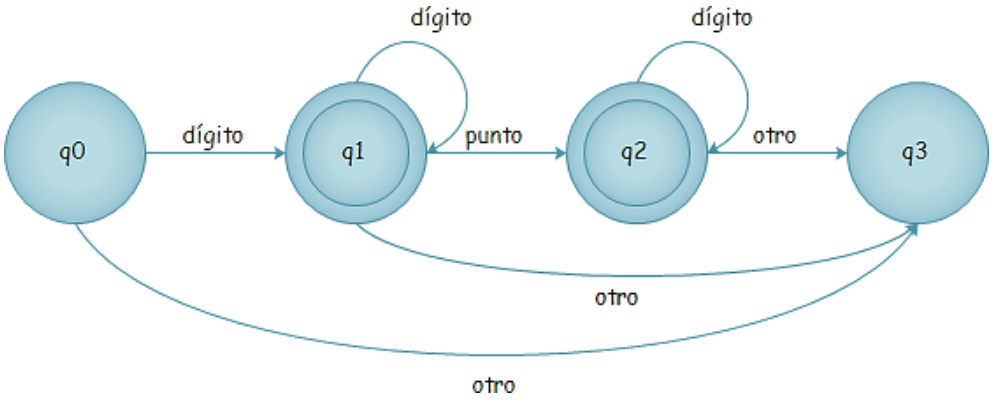


Figura N° 15: DFA para números enteros y reales

Fuente: Elaboración propia

q0: Estado inicial.

q1: Estado de aceptación de un número entero.

q2: Estado de aceptación de un número real.

q3: Estado de rechazo.

- DFA utilizado para reconocer una cadena.

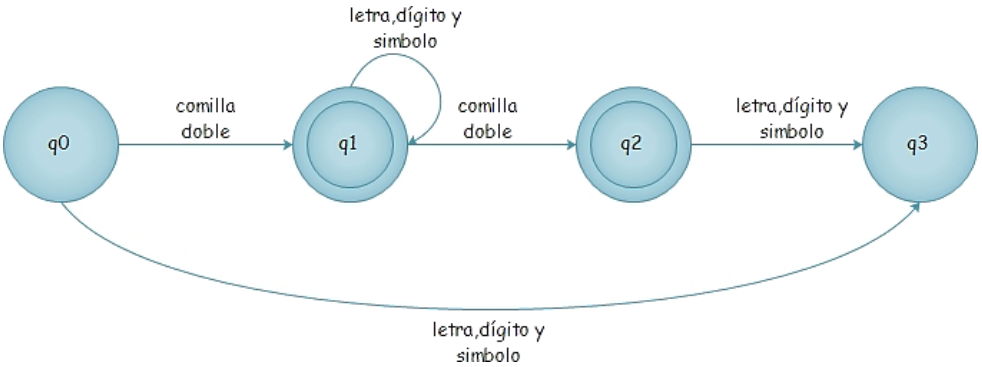


Figura N° 16: DFA para cadenas

Fuente: Elaboración propia

q0: Estado inicial.

q1: Estado de rechazo.

q2: Estado de aceptación de una cadena.

q3: Estado de rechazo.

- DFA utilizado para reconocer un carácter.

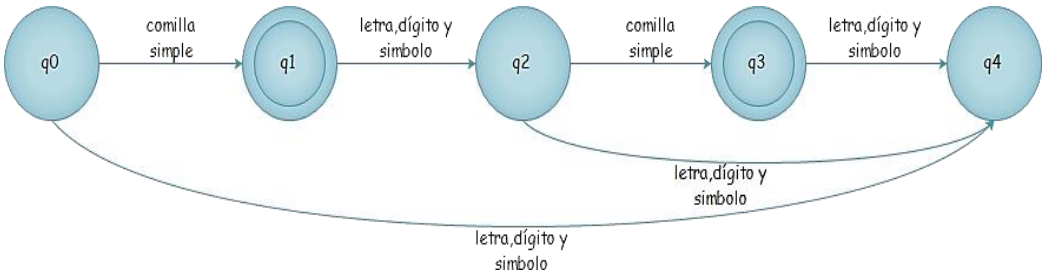


Figura N° 17: DFA para caracteres

Fuente: Elaboración propia

q0: Estado inicial.

q1: Estado de rechazo.

q2: Estado de rechazo.

q3: Estado de aceptación de un carácter.

q4: Estado de rechazo.

E. Clasificación de operadores y símbolos especiales

Los operadores serán clasificados en dos tipos diferentes: los operadores dobles (compuestos por dos símbolos) y los operadores simples y símbolos especiales.

a. Operadores dobles. Son los operadores de mayor o igual (>=), menor o igual (<=), el de igualdad (= =) y el diferente.

b. Operadores simples. Son los operadores asignador (=), suma (+), resta (-), multiplicación (*), división (/), mayor que (>), menor que (<), conjunción (&) y disyunción (|).

Los símbolos especiales que se permiten son: numeral (#), paréntesis de apertura y de cierre, punto y coma, corchete de apertura y de cierre.

5.2.3.2. Analizador Sintáctico

A. Analizador sintáctico descendente

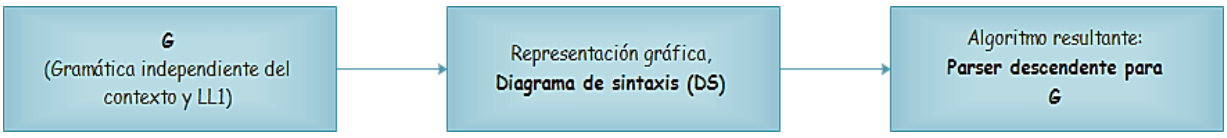


Figura N° 18: Diagrama para la construcción de analizador sintáctico descendente

Fuente: Elaboración propia

La construcción de un analizador sintáctico descendente parte de una gramática independiente del contexto, la cual cumple con ser una gramática LL1, pasando por un diagrama de sintaxis y resultando en un algoritmo, quién es al final implementado en código.

5.2.3.3. Tabla de errores

- Definición de la estrategia a seguir para la presentación de los errores.

  Con el objetivo de simplificar este proceso de presentación de errores en tiempo de compilación al usuario, se ha optado por una estrategia cuyos dos elementos constituyentes son:

  - Un ArrayList que contenga cada una de estas cadenas correspondientes a un posible error generado durante el proceso de compilación de un programa escrito en el lenguaje Hito.
  - Una función encargada de mostrar el error generado y que utilice el ArrayList en base a un número que corresponderá a la posición que contiene el error que se desea mostrar.

  Entonces, cuando sea necesario comunicarle un error al usuario, será suficiente invocar a la función correspondiente enviándole como parámetro un número entero correspondiente a la posición del ArrayList que se desea mostrar, simplificando así este proceso y tornándolo a la vez más práctico y de fácil mantenimiento.

5.2.3.4. Tabla de símbolos

- Definición de elementos a almacenar en la tabla de símbolos.

  Antes de definir la estructura de la tabla de símbolos que será utilizada por las distintas fases del compilador, es necesario definir y tener claro cuáles son los elementos que se desean almacenar dentro de ella.

  En base a la sintaxis del lenguaje HITO (Ver Anexo G), se han identificado los siguientes elementos como necesarios de almacenamiento:

  - El nombre del algoritmo para un archivo determinado que contenga un programa escrito en el lenguaje HITO.
  - Los distintos subalgoritmos que podrían ser definidos para un algoritmo dentro de un programa escrito en el lenguaje HITO.
  - Los parámetros de los subalgoritmos definidos dentro de un programa escrito en el lenguaje HITO. Los cuales se utilizarán como variables dentro del ámbito del subalgoritmo al que pertenecen.

- Las variables definidas y utilizadas, tanto para el algoritmo de un programa escrito en el lenguaje HITO como para todos sus subalgoritmos definidos.

❖ Definición de información necesaria para los objetos a almacenar en la tabla de símbolos.

Como segundo paso es necesario definir para cada tipo de objeto que será almacenado en la tabla de símbolos la información que es necesaria para su manipulación, identificación y ordenamiento.

A continuación se presentan los datos que serán almacenados para cada objeto definido:

- Para el caso de los algoritmos se necesita conocer:
  - Nombre del algoritmo.
- Para el caso de los subalgoritmos se necesita conocer:
  - Nombre del subalgoritmo.
  - Si se trata de una función o un procedimiento.
  - Tipo de dato a retornar por el subalgoritmo si se trata de una función.
  - Cantidad, tipo y estructura de los parámetros.
- Para el caso de los parámetros se necesita conocer:
  - Nombre del parámetro.
  - Tipo de dato del parámetro.
  - Si se trata de un arreglo o una matriz.
- Para el caso de las variables se necesita conocer:
  - Nombre de la variable.
  - Tipo de dato de la variable.
    - ✓ Si se trata de un arreglo o una matriz.
    - ✓ Su longitud (en caso de ser un arreglo o una matriz).
    - ✓ Su anchura (en caso de ser una matriz).
    - ✓ Ámbito al que pertenece y su ubicación dentro de éste.

- Definición de campos de la tabla de símbolos.

  En base a lo expuesto en la sección anterior, se determinó que la tabla de símbolos contará con los siguientes campos:

  - Nombre del objeto (necesario para todos los tipos de objeto que se almacenarán).
  - Tipo del objeto (si se trata de un algoritmo, un subalgoritmo, un parámetro o una variable).
  - Subtipo del objeto (si se trata de una función o un procedimiento para el caso de los subalgoritmos, o si se trata de un arreglo o una matriz para el caso de los parámetros y las variables).
  - Tipo de dato del objeto (para los objetos que necesiten un tipo de dato como es el caso de los parámetros, las funciones y las variables).
  - Longitud de un arreglo o un matriz (para las variables que sean cualquiera de estos tipos de estructura).
  - Anchura de una matriz (para las variables que lo sean).
  - Cantidad de parámetros (para el caso de los subalgoritmos).
  - Valor booleano que indique utilización dentro de una instrucción para (necesario también en el caso de las variables).
  - Nivel del objeto que indique el ámbito para una variable (si se trata de una variable local o global).
  - Dirección del objeto dentro del ámbito al que pertenece (utilizado exclusivamente por variables).
  - La lista de parámetros que el objeto posee (para el caso de los subalgoritmos).

- Definición del tipo de estructura para la tabla de símbolos

  En base al lenguaje HITO, y en vista que se desea implementar un código legible y fácil de interpretar, se optó por diseñar una tabla de símbolos cuyo diseño, organización y sistema de acceso sean de tipo lineal.

  Además, para optimizar un poco la agilidad y eficiencia en el uso de la tabla de símbolos, dicha organización lineal será implementada en el lenguaje C# mediante una ArrayList.

- Definición de elementos necesarios para el almacenamiento, extracción y eliminación de elementos en la tabla de símbolos.

  Para poder contar con un nivel aceptable de eficiencia en el uso de la tabla de símbolos se han considerado como necesarios los siguientes elementos:

  - Un apuntador al primer registro de la tabla de símbolos de forma que se puede utilizar como frontera en los procesos de evaluación y de búsqueda.
  - Un apuntador al último registro añadido a la tabla de símbolos de forma que se pueda contar con su información mientras su proceso de evaluación permanezca activo.
  - Una función encargada de añadir un registro a la tabla de símbolos y de colocar en todos sus campos los datos correspondientes que se consideren necesarios.
  - Una función encargada de buscar dentro de la tabla de símbolos algún registro determinado utilizando como llave de búsqueda el nombre del objeto almacenado.
  - Una función encargada de eliminar registro por registro el contenido de la tabla de símbolos de forma que se pueda liberar todo el espacio en memoria utilizado al momento de terminar el proceso de compilación.

5.2.3.5. Analizador Semántico

El analizador se encarga de validar aspectos que el analizador sintáctico es incapaz de verificar, por ejemplo la comprobación de tipos y comprobación del alcance de las variables utilizadas a lo largo del programa. El desarrollo del analizador semántico del pseudocódigo, está dividido en 3 componentes los cuales recorren el árbol sintáctico durante la etapa de análisis sintáctico.

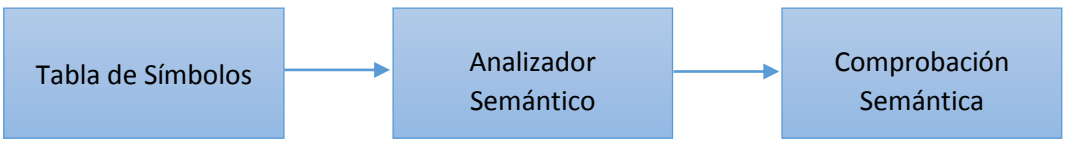
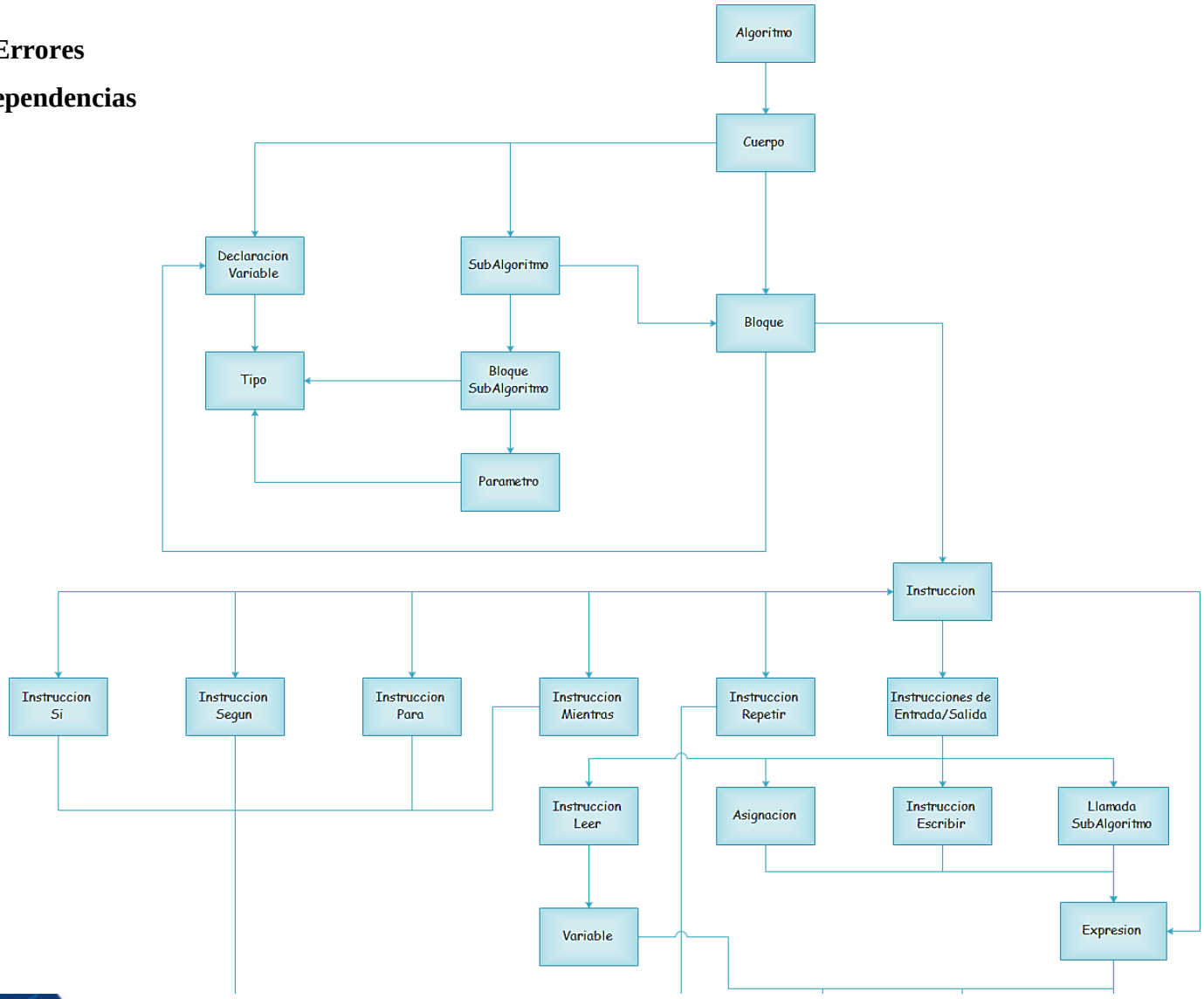


Figura N° 19: Diagrama de flujo del analizador semántico

Fuente: Elaboración propia

5.2.3.6. Tratamiento de Errores

A. Diagrama de dependencias



El diagrama anterior muestra la manera en que las funciones del analizador sintáctico del compilador se llaman unas a otras durante un proceso de compilación. Este diagrama es utilizado para orientar y simplificar la implementación del compilador.

B. Errores sintácticos y semánticos

- Cuando dentro de una función del analizador sintáctico del compilador se haga un llamado a alguna otra función, se enviará como parámetro un conjunto de tokens seguidores, el cual será conformado por todas las palabras claves o caracteres especiales del lenguaje (como paréntesis de cierre, corchetes de cierre o comas) que se esperan ver después de que la función invocada termine de realizar el análisis que le corresponde.

  El objetivo de este conjunto es que ante la presencia de un error en la función invocada (o incluso invocaciones que ella pueda realizar) se logre saltar texto hasta encontrar un token con el que la función (la que realizó la primera invocación de la secuencia) pueda continuar analizando el código fuente.
- La lógica de construcción del compilador permite que en diversos sectores del código al invocar a una función del analizador sintáctico se cuente en efecto con los tokens que esta función necesita para trabajar.
- La regla de las palabras clave indica que para ejecutar los saltos de texto deben utilizarse exclusivamente palabras clave y algunos caracteres de enriquecimiento. Sin embargo, se hará una excepción a esta regla y se incluirá a los identificadores dentro de los criterios de salto en los siguientes casos:
  - ✓ Al inicio y al final de la función encargada de evaluar las instrucciones, dado que de acuerdo a los diagramas de sintaxis siempre puede colocarse una instrucción seguida de otra instrucción, y es necesario tomar en cuenta al identificador para realizar un intento de salto ya que dos tipos de instrucciones pueden comenzar por un identificador (las llamadas a subalgoritmos y las asignaciones).
  - ✓ Al momento de invocar a la función instrucción dentro de la función bloque, de manera que pueda controlarse el inicio de un bloque de instrucciones en los casos en que la primera instrucción del bloque sea una llamada a subalgoritmo o una asignación.

5.2.3.7. Generador binario

El generador binario es un componente del compilador el cual cuenta varias funcionalidades implantadas en el compilador antes de generar el ejecutable.

- ✓ La primera funcionalidad en el generador de código, que traduce el seudocódigo en español al código destino, en este caso código C#. al igual que con los componentes del analizador semántico, el generador de código realiza un recorrido en el árbol sintáctico para traducir cada uno de los nodos.
- ✓ La siguiente funcionalidad es el generador EXE el cual, utiliza el codigo C# generado anteriormente, se genera el ejecutable utilizando la librería System.CodeDom.Compiler.

5.2.3.8. Control de versión

Para el trabajo organizado del proyecto y métodos eficientes de integración de partes del código fuente fue necesario utilizan una herramienta para el control de versiones y mantener un historial de cada cambio introducido sobre cada parte, Para ello se utilizó el software Tortoise SVN que permitió tener un control de las modificaciones en el código fuente del compilador de seudocódigo.

Primero se dio vistazo general a la organización de un proyecto en SVN, se creó un repositorio, donde están todos los archivos (y todas sus versiones) del proyecto. Ese repositorio se puede acceder mediante un cliente SVN y sincronizarse mutuamente.

Con el cliente SVN se realizó principalmente dos grandes acciones: commit y update para guardar y actualizar el código fuente. Existen otras acciones como el checkout, que se utilizó en el proyecto para obtener una copia de todos los archivos del proyecto.

A. Metodología de trabajo

El repositorio donde están los archivos del proyecto y sus versiones se trabajó de manera totalmente local. De este modo, es muchísimo más fácil introducir cambios y probarlos inmediatamente. Incluso se puede trabajar sin conexión a internet.

1. El primer paso fue, hacer un checkout del proyecto, para poder recibir todos los archivos. El checkout se realizó sobre una carpeta o directorio del sistema.
2. Luego se trabaja con archivos locales. Agregar, modificar o eliminar archivos.
3. Cuando se considera que los cambios están lo suficientemente estables, hacemos un commit para que todos los archivos del proyecto puedan tener los cambios incorporados.
4. Para trabajar siempre sobre la última versión del proyecto hacemos un update.

B. Herramientas

➢ El cliente SVN utilizado es el Tortoise SVN, el cual se integra perfectamente al escritorio Windows.

➢ De igual manera se utilizó la versión del cliente SVN para integrar con el IDE de desarrollo de Visual Studio 2012.

C. Historial de versiones

El historial de versiones del compilador de pseudocódigo se inició con el lanzamiento de Hito. La primera versión de Hito 1.0, fue lanzada en octubre de 2015 y se ha publicado varias actualizaciones desde su primer lanzamiento. Estas actualizaciones típicamente corrigen fallos de programa y agregan nuevas funcionalidades.

Tabla N° 9: Historial de versiones

Versión	Fecha lanzamiento	Características
1.0	10 de octubre de 2015	➢ Primera versión oficial de Hito ➢ Herramientas principales de edición de pseudocódigo. ➢ Opciones de compilación(ejecutar código y versificación de sintaxis) ➢ Comando de acceso rápido ➢ Herramientas y ayuda ➢ Ejemplos aplicativos

1.1	26 de octubre de 2015	➢ Corrección de errores en el compilador ➢ Nueva funcionalidad de publicación Local(Ejecutable .exe con icono personalizado)
1.2	20 de noviembre de 2015	➢ Funcionalidad Cloud (Permite realizar publicaciones y descargar de pseudocódigo desde un servidor en internet). ➢ Corrección de algunos debug en el instalador del compilador.
1.3	01 de noviembre de 2015	➢ Funcionalidad de ayuda rápida (Permite mostrar ayuda según el código fuente seleccionado con ejemplos). ➢ Corrección de errores (Permite mostrar el número de línea en el que se encuentre el error)
1.4	18 de diciembre de 2015	➢ Mejora en el código fuente del compilador. ➢ Se modificó la función principal a (Inicio – Fin).
1.5	04 de enero de 2016	➢ Nueva funcionalidad de generación de pseudocódigo(Entrada, proceso y salida) ➢ Mejora en el código fuente del compilador.
2.0	01 de febrero de 2016	➢ Nueva herramienta de generación de gráficos por pseudocódigo. ➢ Actualizaciones del código fuente.

Fuente: Elaboración propia

C. Herramienta Desarrollada

Llevadas a cabo varias iteraciones en el proceso de desarrollo e implementación, se culminó con éxito la herramienta. En el Anexo C se adjunta parte del código fuente desarrollado.

Básicamente, la herramienta construida está constituida por: el editor de pseudocódigo (ver figura 21), el cual permite a los estudiantes la escritura, compilación y ejecución de pseudocódigo.

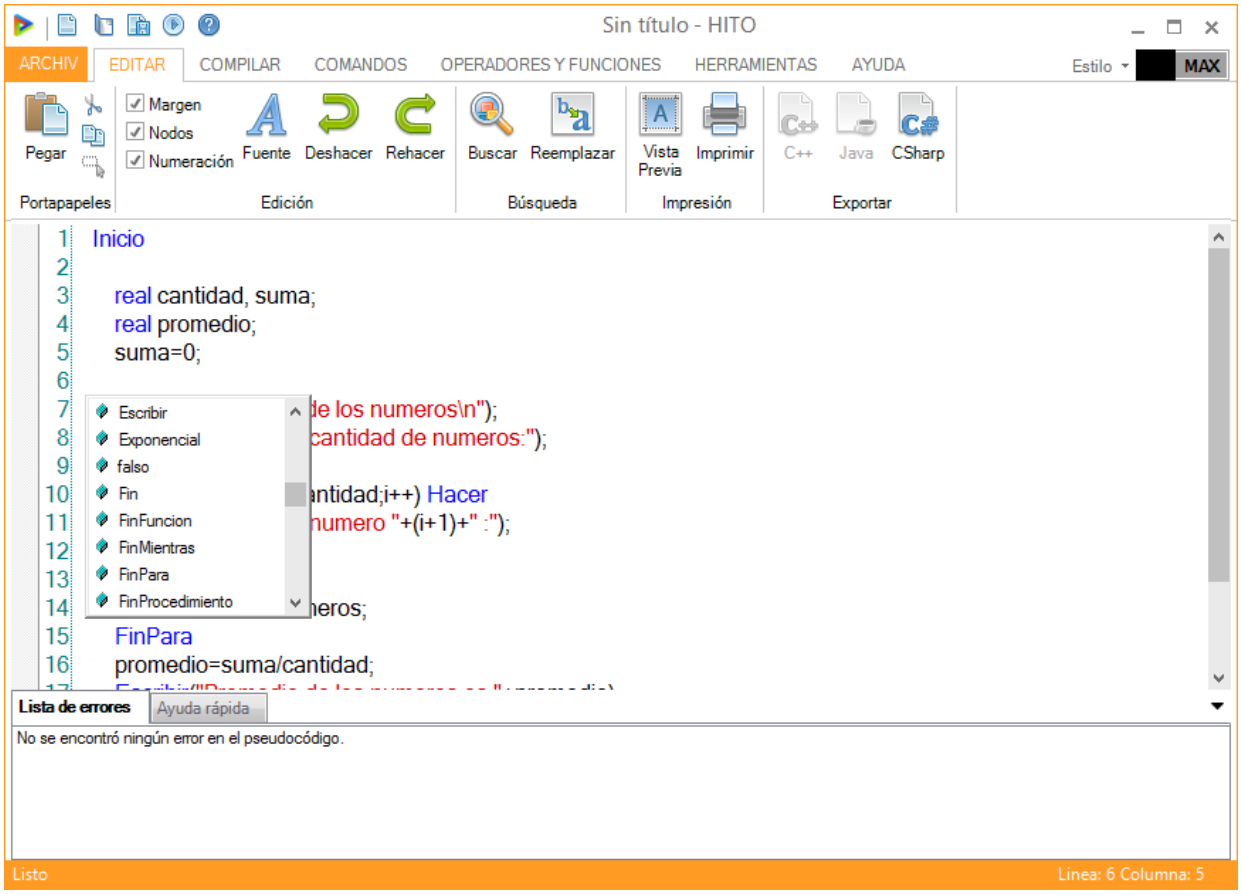


Figura N° 21: Vista de Edición de Pseudocódigo

Fuente: Elaboración propia

El editor de pseudocódigo cuenta con un panel que permite visualizar las líneas de código, y un editor de texto que realiza syntax highlight. Además cuenta con múltiples herramientas, tales como la lista de errores, ayuda rápida que permite mostrar conceptos y ejemplos del código seleccionado, un panel de autocompletado, y una barra de opciones para editar la fuente del editor, un menú de compilación para verificar y ejecutar el código, una barra de comando, operaciones y funciones para agregar código de manera rápita, barra de herramientas y barra de ayuda.

D. Desarrollo del editor de pseudocódigo

El editor de pseudocódigo fue desarrollo utilizando la librería DotNetBar WinForms el cual cuenta con más de 50 contralores de Visual Studio que ayudan a crear la interfaz de usuario profesional con facilidad. Entre los controles utilizados están Office 2010 Ribbon Control, Office 2007, 2003 y menús y barras de herramientas de estilo VS.NET 2005, los controles del panel multi-funcionales, los controles de acoplamiento, Panel de navegación, etc. En el anexo F se adjunta algunos pseudocódigos creados con el compilador.

El editor de pseudocódigo permite administrar estilos y colores para todos los controles de la interfaz de usuario como se muestra en las siguientes figuras:

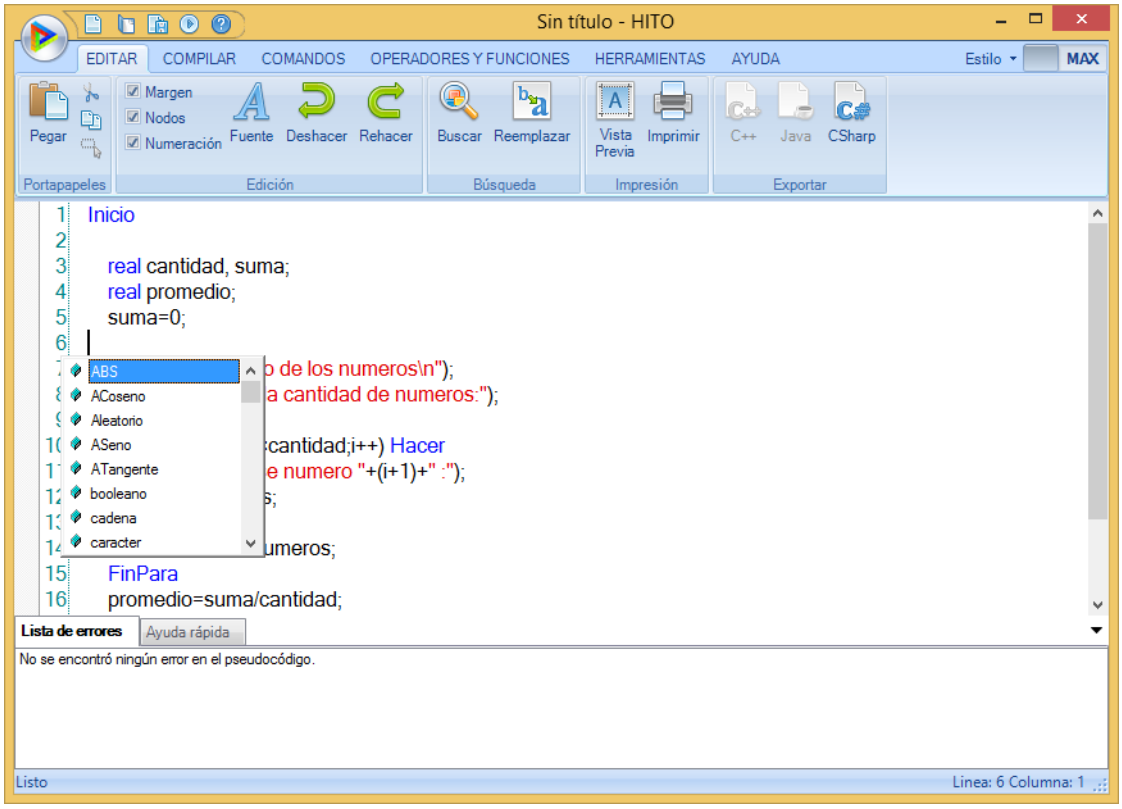


Figura N° 22: Estilo Office 2007 blue

Fuente: Elaboración propia

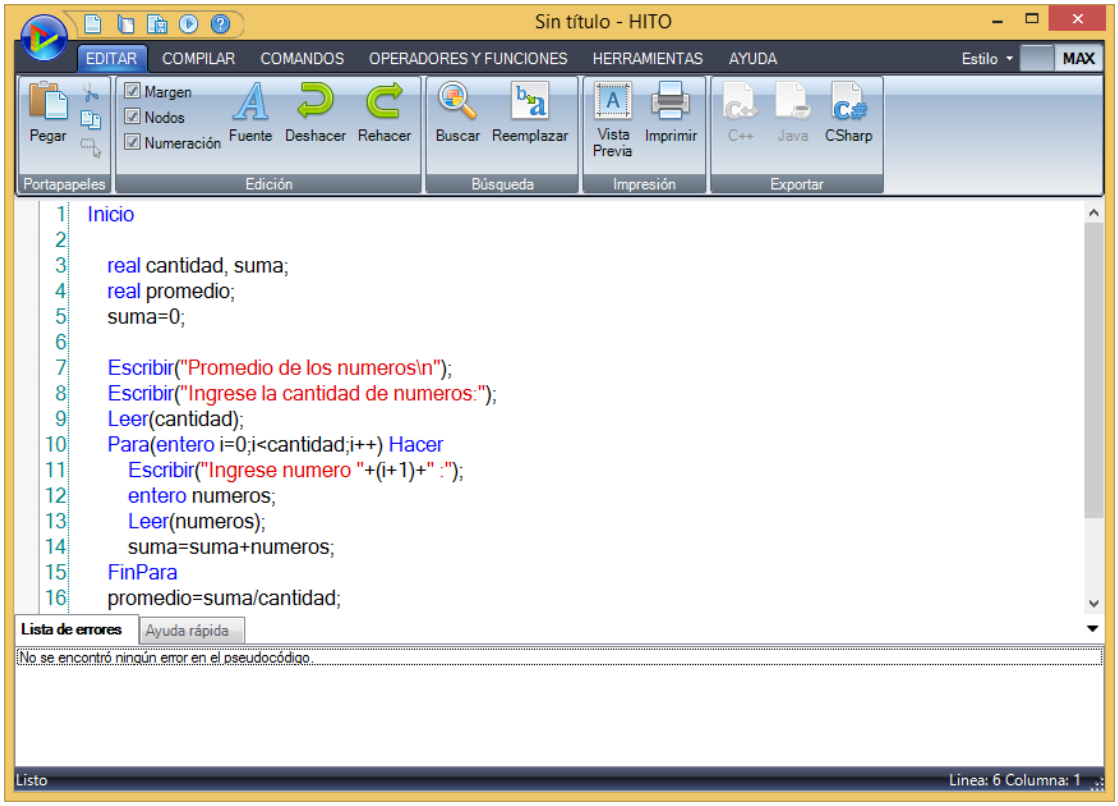


Figura N° 23: Estilo Vista glass

Fuente: Elaboración propia

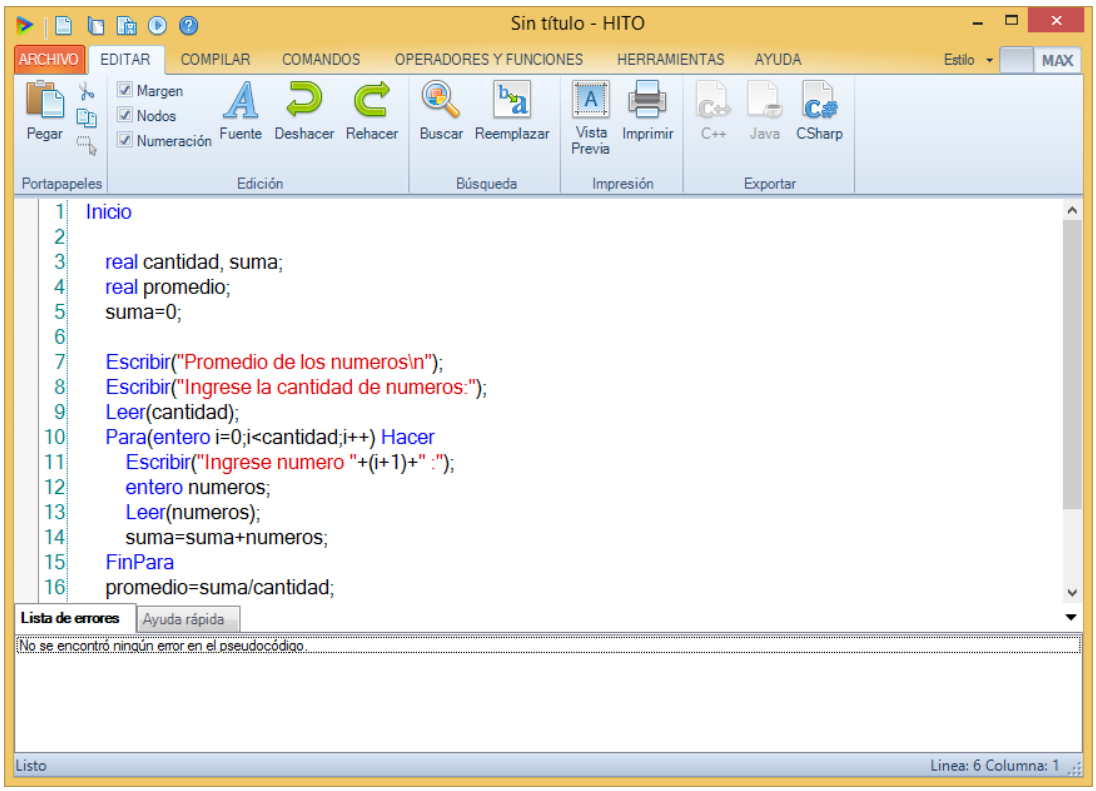


Figura N° 24: Estilo Office 2010 blue

Fuente: Elaboración propia

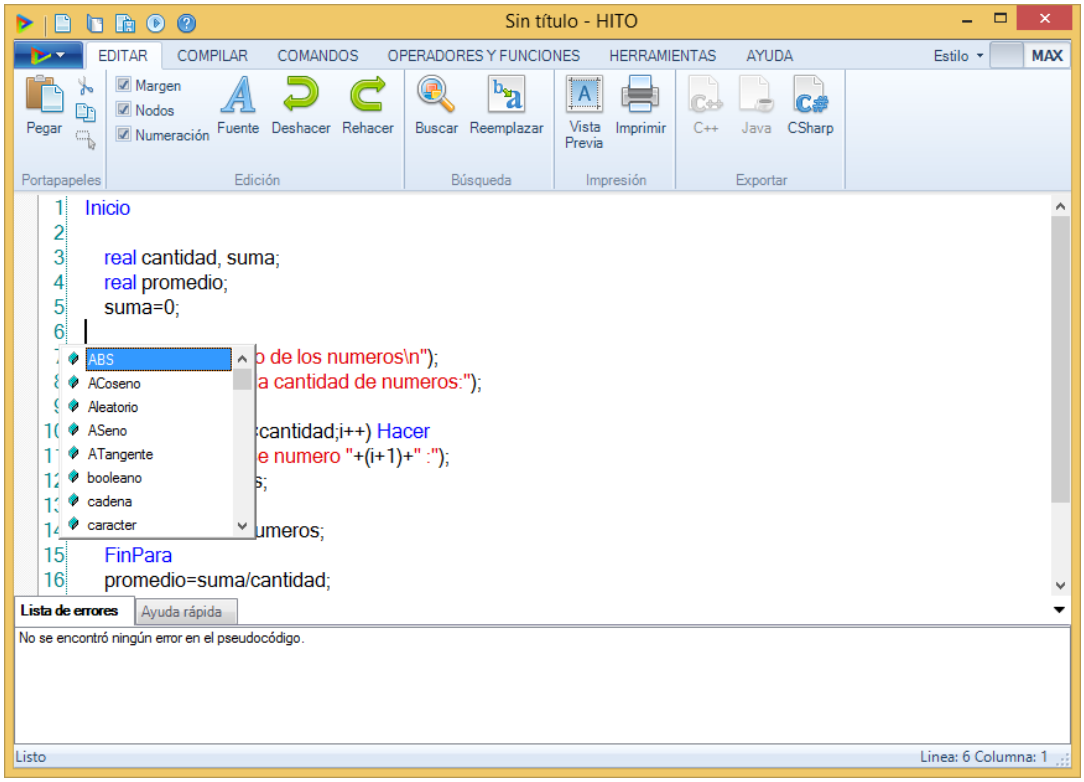


Figura N° 25: Estilo Windows 7

Fuente: Elaboración propia

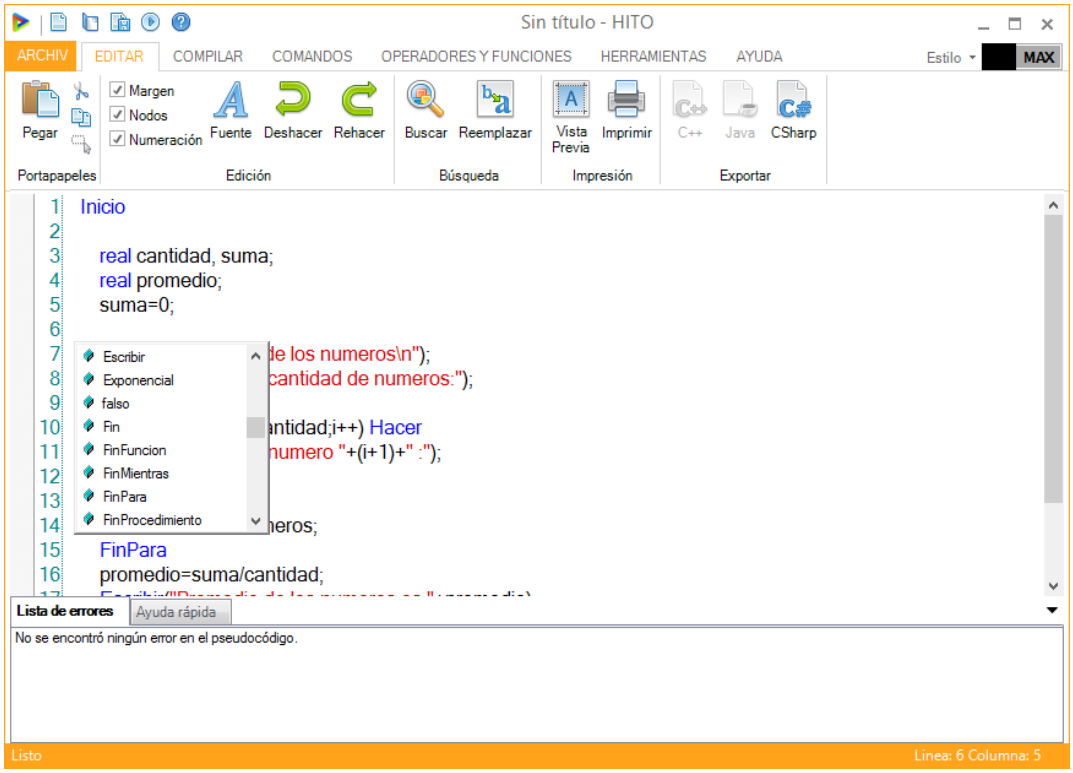


Figura N° 26: Estilo Metro

Fuente: Elaboración propia

5.2.3.9. Publicación Local y Cloud

Una de las funcionalidades extras del compilador es la de poder publicar los pseudocódigos en un servidor en internet, la ubicación en el servidor web permitirá todos los que usen la herramienta puedan descargar libremente los pseudocódigo publicados.

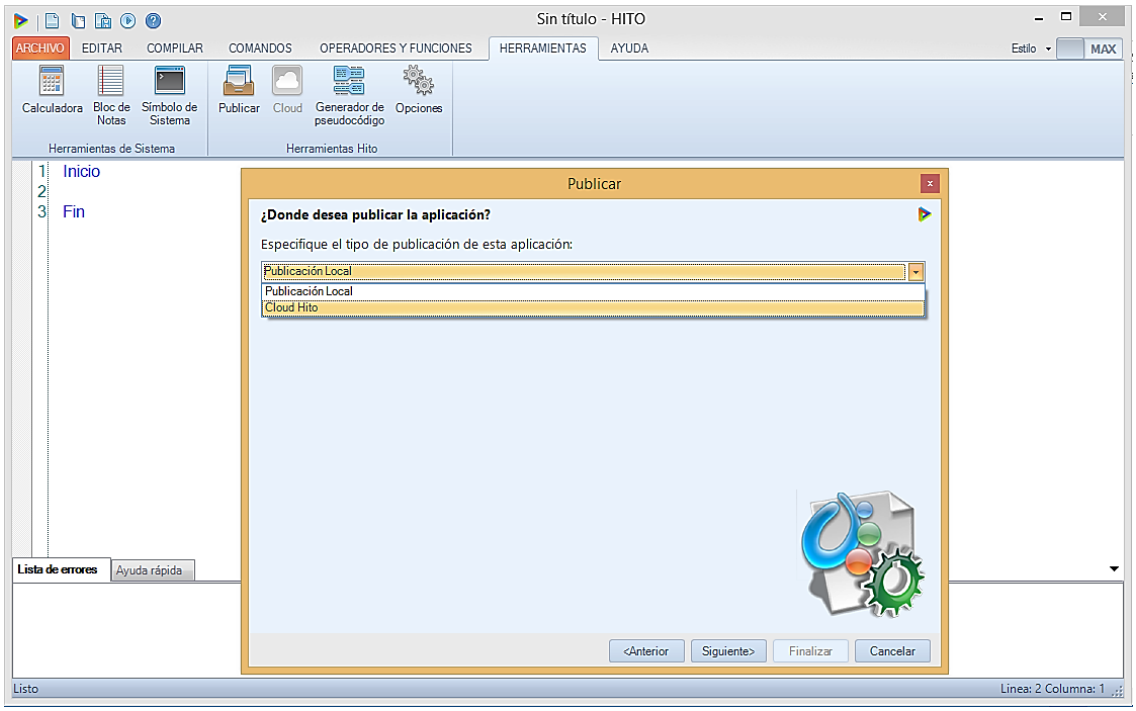


Figura N° 27: Herramienta de publicación de pseudocódigo

Fuente: Elaboración propia

5.2.3.10. Generador de Pseudocódigo

Esta funcionalidad del compilador permite generar un pseudocódigo utilizando una interfaz de usuario intuitiva siguiente los pasos de: entrada, proceso y salida del cual todo algoritmo está basado. Permite que los alumnos diferencien cada una de las fases de creación de un pseudocódigo.

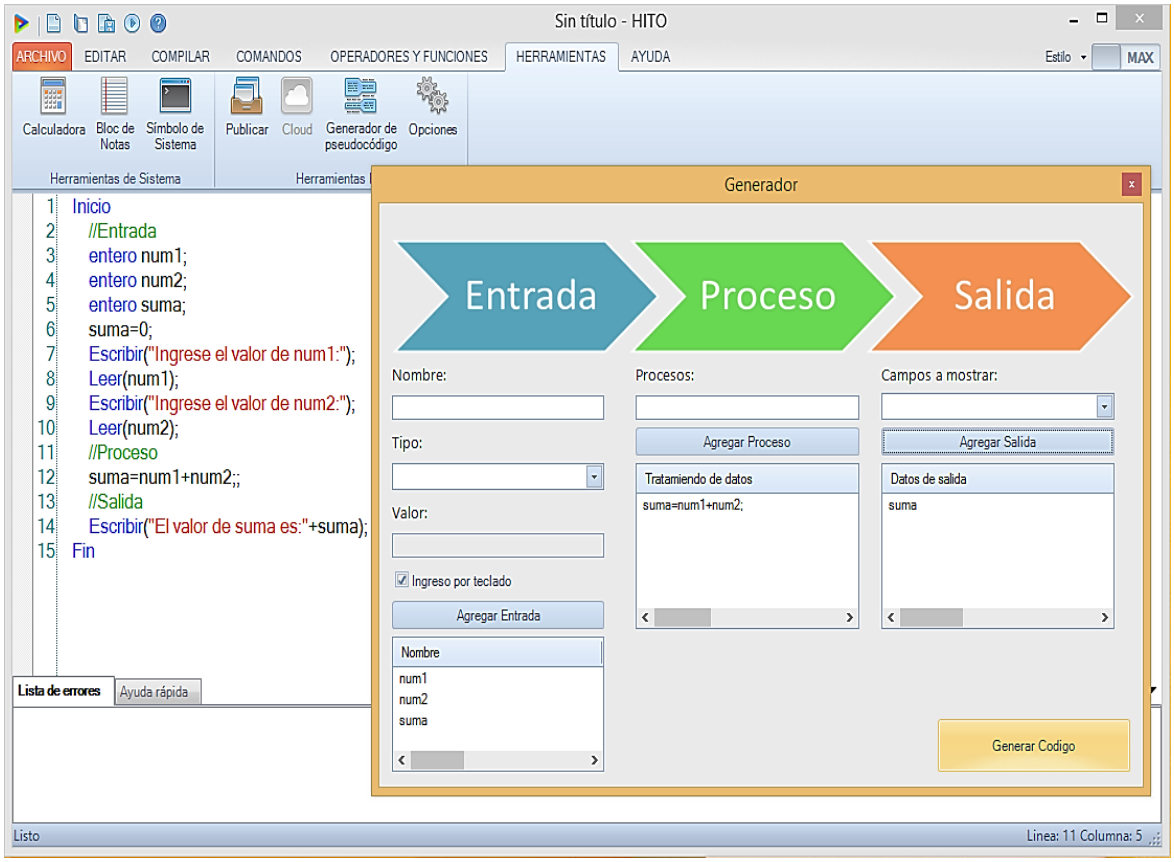


Figura N° 28: Herramienta de generador de pseudocódigo

Fuente: Elaboración propia

5.2.3.11. Gráficos

Los comandos de gráficos implementados en el compilador permiten generar graficas desde el pseudocódigo. Los comandos de graficas disponibles son: DibujarPunto, DibujarLinea, DibujarImagen, DibujarTexto, DibujarRectangulo, etc.

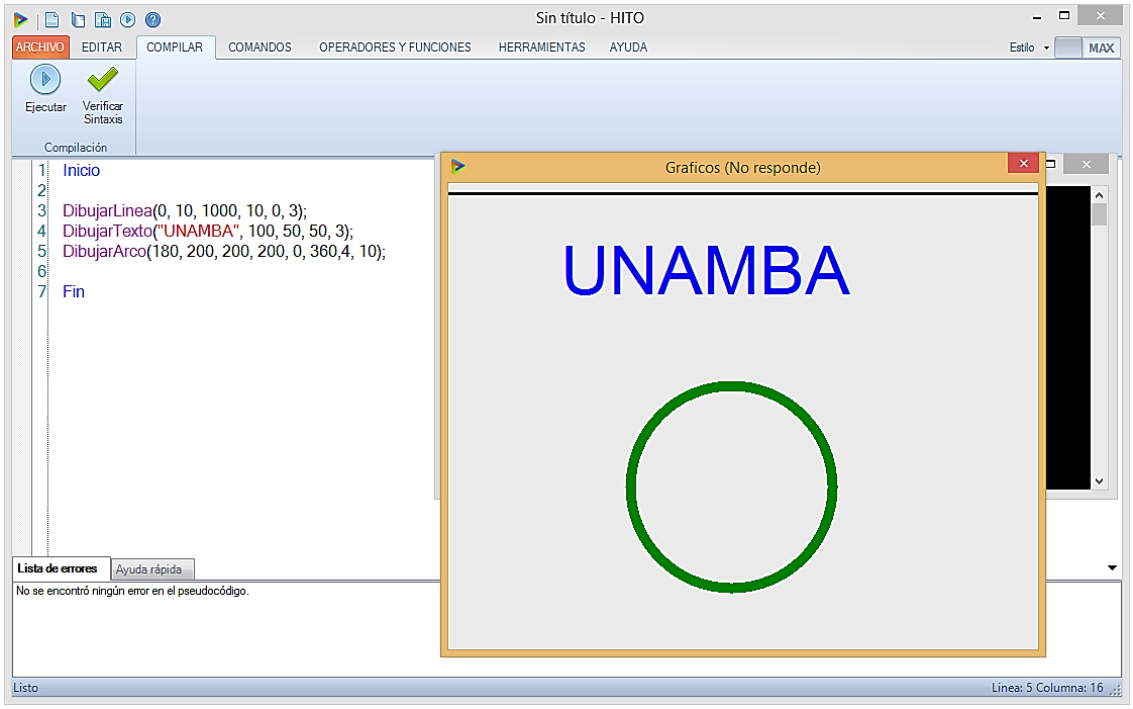


Figura N° 29: Ventana grafica de la librería grafica del compilador

Fuente: Elaboración propia

5.2.4. Pruebas

Una vez finalizada cada iteración de desarrollo, se ejecutaron las pruebas pertinentes para verificar la trazabilidad y la funcionalidad de la herramienta.

5.2.4.1. Pruebas Unitarias

Se llevó a cabo un conjunto de pruebas por cada componente, verificando y validando el correcto funcionamiento del mismo. En caso de que el componente no pasara la prueba, se hizo una reiteración, analizando los aspectos en donde la implementación fallaba, para corregirla y seguir con el proceso de revalidación y verificación (Anexo D).

5.2.4.2. Pruebas de Sistema

Con las pruebas unitarias e integrales realizadas, los componentes validados y verificados se sometieron a pruebas de sistema, para asegurar que la herramienta funcionara y cumpliera con los requerimientos funcionales y no funcionales extraídos de los User Stories enunciados anteriormente. Se ejecutaron pruebas tales como la creación, edición, compilación y ejecución de archivos de Pseudocódigo, provistos en el Anexo D.

CONCLUSIONES

Después de culminar con el trabajo de investigación, se llegaron a las siguientes conclusiones:

- El compilador de pseudocódigo denominado HITO contribuye significativamente en la capacidad procedimental en el desarrollo de modularidad. Según la prueba de hipótesis aplicada y con un nivel de significancia de 5%, se obtuvo el siguiente resultado: De acuerdo a los dos grupos de muestra aplicando la distribución t de Student se obtiene un Tc= 3.9060 y un Tt=1.6820. Al obtener un Tc > Tt se rechaza la hipótesis nula y se acepta la hipótesis alterna, por lo que podemos afirmar que el compilador de pseudocódigo si contribuyó al aprendizaje en el desarrollo de modularidad en la asignatura de Algorítmica I.
- El compilador de pseudocódigo denominado HITO contribuye significativamente en la capacidad procedimental en el desarrollo de procedimientos. Según la prueba de hipótesis aplicada y con un nivel de significancia de 5%, se obtuvo el siguiente resultado: De acuerdo a los dos grupos de muestra aplicando la distribución t de Student se obtiene un Tc= 2.2504 y un Tt= 1.6820. Al obtener un Tc > Tt se rechaza la hipótesis nula y se acepta la hipótesis alterna, por lo que podemos afirmar que el compilador de pseudocódigo si contribuyó al aprendizaje en el desarrollo de procedimientos en la asignatura de Algorítmica I.
- El compilador de pseudocódigo denominado HITO contribuye significativamente en la capacidad procedimental en el desarrollo de funciones. Según la prueba de hipótesis aplicada y con un nivel de significancia de 5%, se obtuvo el siguiente resultado: De acuerdo a los dos grupos de muestra aplicando la distribución t de Student se obtiene un Tc= 2.6814 un Tt= 1.6820. Al obtener un Tc > Tt se rechaza la hipótesis nula y se acepta la hipótesis alterna, por lo que podemos afirmar que el compilador de pseudocódigo si contribuyó al aprendizaje en el desarrollo de funciones en la asignatura de Algorítmica I.
- Se desarrolló una herramienta para la enseñanza de la programación, un compilador funcional de pseudocódigos y todos sus componentes.
- El GUI del compilador se desarrolló con la librería DotNetBar Win Forms el cual permitió crear una aplicación profesional e intuitiva.

- Se incorporó al compilador la librería gráfica System.Drawing, el cual permite generar las gráficas primitivas.
- Se concluye que el compilador de pseudocódigo en español facilito a los estudiantes el aprendizaje y el desarrollo modular de pseudocódigos.

RECOMENDACIONES

A continuación se plantean las recomendaciones para futuros trabajos que no pudieron ser implementadas en el trabajo de investigación:

- El lenguaje desarrollado para el pseudocódigo sólo maneja datos básicos y estos necesitan ser declarados. Debido a esto un posible trabajo futuro sería incluir la capacidad de definir estructuras de dato, lo cual aumentaría la dificultad al momento de la traducción y sería necesario definir estos nuevos tipos de datos para evitar dificultades al detectar e identificar los tipos de datos.
- El entorno de desarrollo actual permite ejecutar pseudocódigo, sería interesante en un posible trabajo futuro aumentar la funcionalidad de asociar compiladores con el entorno para que se permita la ejecución de otros lenguajes de programación.
- Se podría aprovechar la capacidad de traducción de código, debido a que se manejan tabulaciones al traducir para una mejor visualización, para a partir del pseudocódigo poder diseñar el diagrama de flujo. Esto vendría a ser una traducción de acciones a figuras y de tabulaciones a espaciado en el diagrama.
- Sería posible extender la traducción de la herramienta al lenguaje VB teniendo en cuenta las limitaciones en la lectura de datos, debido a que en C# no se puede leer más de 2 valores en una misma línea con una sola sentencia. En este caso se recomendaría leer por cada línea un solo valor.
- Finalmente, un posible trabajo futuro podría ser implementar la etapa de optimización de código para acelerar la ejecución del código de programas en pseudocódigo.

BIBLIOGRAFÍA

- BALTASAR GARCÍA, Perez-Schofield (2012). Introducción a la programación con jC.
- BRASSARD, GILLES; BRATLEY, Paul (1997). Fundamentos de Algoritmia. Madrid: PRENTICE HALL.
- CANALES MARTÍNEZ, Isaac Andrés y RUIZ TEJEIDA Michel (2011). Desarrollo de un Compilador para Pseudocódigo en Lenguaje Español. Instituto Politécnico Nacional - Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas. México.
- GALVEZ ROJAS, S., & Mora Mata, M. A. (2005). Compiladores: Traductores y compiladores con Lex/Yacc, Jflex/Cup y JavaCC. Malaga, Universidad de Malaga. España.
- International Organization for Standardization ISO 9241, "Ergonomic requirements for office work with visual diplay terminals".
- JARA LOAYZA, Juan Carlos (2013). Entorno de Desarrollo para la Ejecución y Traducción de Pseudocódigo. Pontificia Universidad Católica del Perú - Facultad de Ciencias e Ingeniería. Perú.
- JOYANES, Luis (2008). Fundamentos de programación: Algoritmos, estructuras de datos y objetos. Cuarta edición. España: McGraw-Hill.
- JOSKOWICZ, José (2006). Reglas y Prácticas en eXtreme Programming.
- KNUTH Donald, E (1964). Backus Normal Form vs. Backus Naur Form. EE. UU.
- LAÍNEZ FUENTES, José Rubén (2015). Desarrolló de software ágil. IT Campus Academy.
- MUÑOZ MARRÓN Elena, PERIÁÑEZ MORALES José Antonio (2012). Fundamentos del aprendizaje y del lenguaje. Editorial UOC.
- PÁEZ PÉREZ, Liliam Paola y VÁSQUEZ ACERO, John Henry (2008). PsiCoder: Software Educativo para Facilitar el Proceso de Enseñanza.Pontificia Universidad Javeriana. Facultad de Ingeniería - Carrera de Ingeniería de Sistemas. Colombia.
- RUIZ CATALÁN, Jacinto (2010). Compiladores – Teoría e implementación. España.
- SANCHEZ DUEÑAS Y VALVERDE, Andreu (1989). Compiladores e Intérpretes un Enfoque Pragmático. Madrid.
- SERGIO GÁLVEZ, R. y Miguel A. MORA M. (2005). Java a Tope: Traductores y Compiladores con LEX/YACC, JFlex/CUP y JavaCC.
- SOMMERVILLE, Ian (2005). Ingeniería del software. Pearson Educación.
- VEGA CASTRO, Rafael Aníbal (2008). Compilador de Pseudocódigo como Herramienta para el Aprendizaje en la Construcción de Algoritmos. Universidad del Norte - Programa de Ingeniería de Sistemas. Colombia.

ANEXOS

ANEXO A

Acta de notas de la asignatura de Algorítmica I del Semestre Académico 2015-I

UNIVERSIDAD NACIONAL MICAELA BASTIDAS DE APURÍMAC

VICERRECTORADO ACADÉMICO

DIRECCIÓN DE SERVICIOS ACADÉMICOS – OFICINA DE REGISTRO Y ARCHIVOS

Escuela Académico Profesional: INGENIERÍA INFORMÁTICA Y SISTEMAS
Semestre: 2015-1
Asignatura: Algorítmica I
Docente: MAMANI VILCA Ecler

Código de Asignatura: IS201
Creditaje: 5
Categoría: AFPO
Grupo: A

REGISTRO DE EVALUACIÓN

N°	Código Estudiante	Apellidos y Nombres	Capacidad																										Actitud								
			Contenido Conceptual													Contenido Procedimental													Contenido Actitudinal								
			I				II				III				PCC	I				II				III				PCP	I				II				PCA
			1	2	3	PI	1	2	3	PII	1	2	3	PIII		1	2	3	PI	1	2	3	PII	1	2	3	PIII		1	2	3	PI	1	2	3	PII	
1	131111	CARRASCO LUPINTA IVAN ELIAS	18	10	10	13									13	20	18	18	19									19	12	15		14					14
2	142149	CHOQUE LAUPA NISBER	10	11	11	11									11	10	10	04	08									08	12	15		14					14
3	141155	COSTILLO ORTIZ YAQUELIN RUTH	10	20	20	17									17	15	06	10	10									10	13	15		14					14
4	141159	GONZALES TINCO DAVID	10	10	04	08									08	05	10	10	08									08	15	15		15					15
5	142158	GUERRERO CHICLLA TOMAS ANDERSON	09	04	04	06									06	10	10	15	12									12	13	15		14					14
6	121164	HILARES HURTADO GIL SABINO	02	10	10	07									07	0	10	14	08									08	14	15		15					15
7	131121	HUACCALSAICO TORRES JOSE OLIVER	14	10	10	11									11	18	20	20	19									19	13	15		14					14
8	141164	MAMANI HURTADO NELSON EVER	06	07	07	07									07	18	13	17	16									16	12	15		14					14
9	141167	MORAYA GUZMAN JUANA ANALI	03	06	06	05									05	10	05	09	08									08	12	15		14					14
10	141168	MOREANO AGUIRRE LUCIA	17	14	14	15									15	18	15	15	16									16	15	15		15					15
11	142175	PAUCAR CAMPOSANO TRENIDAD	06	04	04	05									05	10	10	04	08									08	15	15		15					15
12	141175	RAMOS BAUTISTA JESUS DENNIS VALERY	02	NSP	NSP	01									01	18	13	17	16									16	15	15		15					15
13	141176	REINOSO TORBISCO JEAN CARLOS	12	07	07	09									09	10	10	14	11									11	18	18		18					18
14	141177	RIOS HUACHACA VIRGILIO ALEXANDER	12	14	14	13									13	20	10	14	15									15	16	15		16					16
15	122190	TRUJILLO CARBAJAL DAVID	10	13	13	12									12	13	05	05	08									08	13	15		14					14
16	141189	VERA CCAHUANA SHEYLA	11	05	05	07									07	20	10	14	15									15	13	15		14					14

ITEM DE EVALUACIÓN								
CAPACIDAD						ACTITUD		
CC	CONTENIDO CONCEPTUAL		CP	CONTENIDO PROCEDIMENTAL		CA	CONTENIDO ACTITUDINAL	
	ITEM	EVAL		ITEM	EVAL		ITEM	EVAL
I	A	3	I	G	3	I	A	2
II			II			II		
III			III					

CAPACIDAD		ACTITUD
CONTENIDO CONCEPTUAL	CONTENIDO PROCEDIMENTAL	CONTENIDO ACTITUDINAL
A.-Pruebas Escritas	G.-Demostración de procesos	A.-Responde a las normas de convivencia

N°	Código Estudiante	RESUMEN NOTAS					Promedio Final	NOTA FINAL	
		PCC	Sust.	PFCC	PCP	PCA		En Nros.	En letras
1	131111	13		13	19	14	16	16	Dieciseis
2	142149	11	12	12	08	14	10	11	Once
3	141155	17		17	10	14	14	14	Catorce
4	141159	08		08	08	15	09	09	Nueve
5	142158	06		06	12	14	09	09	Nueve
6	121164	07		07	08	15	08	08	Ocho
7	131121	11		11	19	14	15	15	Quince
8	141164	07		07	16	14	11	11	Once
9	141167	05		05	08	14	07	07	Siete
10	141168	15		15	16	15	15	15	Quince
11	142175	05		05	08	15	07	07	Siete
12	141175	01		01	16	15	08	08	Ocho
13	141176	09		09	11	18	11	11	Once
14	141177	13		13	15	16	14	14	Catorce
15	122190	12		12	08	14	11	11	Once
16	141189	07		07	15	14	11	11	Once

Observaciones

CONSOLIDADO	En Nros
Estudiantes aprobados	10
Estudiantes desaprobados	4
Estudiantes reprobados	2
Estudiantes NSP	0
Otros	
Total de Estudiantes matriculados	16

LEYENDA	
PI, PII, PIII	Promedio de Sub – contenido Conceptual, Procedimental, Actitudinal
PCC	Promedio de Contenido Conceptual
PCP	Promedio de Contenido Procedimental
PCA	Promedio de Contenido Actitudinal
PFCC	Promedio final de Contenido Conceptual
Sust.	Sustitutorio
CC, CP, CA	Nro. de Contenido Conceptual, Procedimental y Actitudinal

Fórmula del PCC	Promedio Aritmético
Fórmula del PCP	**Promedio Aritmético**
Fórmula del PCA	**Promedio Aritmético**

Fórmula de la Nota Final (La fórmula del promedio final es la misma de la nota final)

$$NF = \frac{PCC\,PFCC(\ 0{,}5\ ) + PCP(\ 0{,}4\ ) + PCA(\ 0{,}1\ )}{(\ 1{,}0\ )}$$

Los pesos son relativos a la naturaleza de las competencias Art. N°18,20,21 del Reglamento de Evaluación y aprendizaje.

Especificaciones sobre sustitutorio
Remitirse al reglamento de Evaluación del Aprendizaje (Resolución N°178-2010-CU-UNAMBA)

Abancay, 07 de Septiembre del 2015

UNIVERSIDAD NACIONAL MICAELA BASTIDAS

ANEXO B

Acta de notas de la asignatura de Algorítmica I del Semestre Académico 2015-II

UNIVERSIDAD NACIONAL MICAELA BASTIDAS DE APURÍMAC

VICERRECTORADO ACADÉMICO

DIRECCIÓN DE SERVICIOS ACADÉMICOS – OFICINA DE REGISTRO Y ARCHIVOS

Escuela Académico Profesional: INGENIERÍA INFORMÁTICA Y SISTEMAS
Semestre: 2015-2
Asignatura: Algorítmica I
Docente: MAMANI VILCA Ecler

Código de Asignatura: IS201
Creditaje: 5
Categoría:
Grupo: A

REGISTRO DE EVALUACIÓN

N°	Código Estudiante	Apellidos y Nombres	Capacidad																										Actitud								
			Contenido Conceptual													Contenido Procedimental													Contenido Actitudinal								
			I				II				III				PCC	I				II				III				PCP	I				II				PCA
			1	2	3	PI	1	2	3	PII	1	2	3	PIII		1	2	3	PI	1	2	3	PII	1	2	3	PIII		1	2	3	PI	1	2	3	PII	
1	142144	AIQUIPA ARBIETO GAVY VALERIA	05	04	05	05									05	05	08	12	08									08	18	17		18					18
2	151139	ALATA VALDERRAMA KEVIN	09	08	10	09									09	05	13	14	11									11	18	17		18					18
3	151160	AVENDAÑO PUMAPILLO DAVID SANTOS	13	13	12	13									13	20	08	12	13									13	18	17		18					18
4	151174	CHAVEZ QUISPE JUAN CRISTIAN	NSP	NSP	NSP	NSP									NSP	NSP	NSP	NSP	NSP									NSP	NSP	NSP		NSP					NSP
5	151152	COLLADO VALENZUELA ROLDAN JIM	20	20	18	19									19	20	18	20	19									19	18	17		18					18
6	142151	CONTRERAS CCAHUANA NOE	09	06	17	11									11	20	08	15	14									14	18	17		18					18
7	142155	DAVALOS JARA ADOLFO	02	04	10	05									05	NSP	NSP	10	03									03	12	13		13					13
8	131119	FUENTES ANDRADE ROBERTO JESUS	08	10	10	09									09	10	13	12	12									12	12	13		13					13
9	141159	GONZALES TINCO DAVID	07	07	15	10									10	10	11	12	11									11	13	13		13					13
10	142158	GUERRERO CHICLLA TOMAS ANDERSON	20	08	15	14									14	20	18	17	18									18	16	17		17					17
11	142159	GUTIERREZ SANCHEZ JOSE ANTONIO	12	15	15	14									14	20	13	15	16									16	18	17		18					18
12	122166	HERRERA ESPINOZA ROCIO LIZBENIA	08	NSP	NSP	03									03	08	NSP	NSP	03									03	12	nsp		06					06
13	151155	IGNACIO TAYPE AQUILINO JONATHAN	08	14	10	11									11	20	12	15	16									16	18	17		18					18
14	131126	MARTINEZ PALOMINO JESUS ANTHONY	07	07	12	09									09	10	10	12	11									11	18	17		18					18
15	141167	MORAYA GUZMAN JUANA ANALI	07	11	11	10									10	15	07	12	11									11	13	14		14					14
16	151173	MOREANO CONDORCUYA LISZETH	09	08	15	11									11	05	08	12	08									08	18	17		18					18
17	142171	OLIVERA CONTRERAS JESUS	08	10	15	11									11	05	12	16	11									11	18	17		18					18
18	142175	PAUCAR CAMPOSANO TRENIDAD	05	10	15	10									10	10	08	13	10									10	18	17		18					18

UNIVERSIDAD NACIONAL MICAELA BASTIDAS DE APURÍMAC

VICERRECTORADO ACADÉMICO

DIRECCIÓN DE SERVICIOS ACADÉMICOS – OFICINA DE REGISTRO Y ARCHIVOS

Escuela Académico Profesional: INGENIERÍA INFORMÁTICA Y SISTEMAS
Semestre: 2015-2
Asignatura: Algorítmica I
Docente: MAMANI VILCA Ecler

Código de Asignatura: IS201
Creditaje: 5
Categoría:
Grupo: A

REGISTRO DE EVALUACIÓN

N°	Código Estudiante	Apellidos y Nombres	Capacidad																									Actitud								
			Contenido Conceptual													Contenido Procedimental													Contenido Actitudinal								
			I				II				III				PCC	I				II				III				PCP	I				II				PCA
			1	2	3	PI	1	2	3	PII	1	2	3	PIII		1	2	3	PI	1	2	3	PII	1	2	3	PIII		1	2	3	PI	1	2	3	PII	
26	151153	VALENZUELA AGUILAR YEIN VICTOR	06	07	10	08									08	10	nsp	12	07									07	18	17		18					18
27	142187	VARGAS MENDOZA IVAN AUGUSTO	09	09	12	10									10	20	12	12	15									15	13	15		14					14
28	151158	VEGA LOAYZA CARMEN ROSA	NSP	nsp	nsp	NSP									NSP	NSP	nsp	nsp	NSP									NSP	nsp	nsp		NSP					NSP

ITEM DE EVALUACIÓN								
CAPACIDAD						ACTITUD		
CC	CONTENIDO CONCEPTUAL		CP	CONTENIDO PROCEDIMENTAL		CA	CONTENIDO ACTITUDINAL	
	ITEM	EVAL		ITEM	EVAL		ITEM	EVAL
I	A	3	I	G	3	I	A	2
II			II			II		
III			III					

CAPACIDAD		ACTITUD
CONTENIDO CONCEPTUAL	CONTENIDO PROCEDIMENTAL	CONTENIDO ACTITUDINAL
A.-Pruebas Escritas	G.-Demostración de procesos	A.-Responde a las normas de convivencia

N°	Código Estudiante	RESUMEN NOTAS					Promedio Final	NOTA FINAL	
		PCC	Sust.	PFCC	PCP	PCA		En Nros.	En letras
1	142144	05		05	08	18	08	08	Ocho
2	151139	09		09	11	18	11	11	Once
3	151160	13		13	13	18	14	14	Catorce
4	151174	NSP		NSP	NSP	NSP	NSP	NSP	No Se Presentó
5	151152	19		19	19	18	19	19	Diecinueve
6	142151	11		11	14	18	13	13	Trece
7	142155	05		05	03	13	05	05	Cinco
8	131119	09		09	12	13	11	11	Once
9	141159	10		10	11	13	11	11	Once
10	142158	14		14	18	17	16	16	Dieciseis
11	142159	14		14	16	18	15	15	Quince
12	122166	03		03	03	06	03	03	Tres
13	151155	11		11	16	18	14	14	Catorce
14	131126	09		09	11	18	11	11	Once
15	141167	10		10	11	14	11	11	Once
16	151173	11		11	08	18	11	11	Once
17	142171	11		11	11	18	12	12	Doce
18	142175	10		10	10	18	11	11	Once
19	142177	10		10	11	18	11	11	Once
20	151170	07		07	05	18	07	07	Siete
21	142181	09		09	12	18	11	11	Once
22	141175	07		07	07	15	08	08	Ocho

Observaciones

CONSOLIDADO	En Nros
Estudiantes aprobados	18
Estudiantes desaprobados	3
Estudiantes reprobados	4
Estudiantes NSP	3
Otros	
Total de Estudiantes matriculados	28

LEYENDA	
PI, PII, PIII	Promedio de Sub – contenido Conceptual, Procedimental, Actitudinal
PCC	Promedio de Contenido Conceptual
PCP	Promedio de Contenido Procedimental
PCA	Promedio de Contenido Actitudinal
PFCC	Promedio final de Contenido Conceptual
Sust.	Sustitutorio
CC, CP, CA	Nro. de Contenido Conceptual, Procedimental y Actitudinal

Fórmula del PCC	Promedio Aritmetico
Fórmula del PCP	**Promedio Aritmetico**
Fórmula del PCA	**Promedio Aritmetico**

Fórmula de la Nota Final (La fórmula del promedio final es la misma de la nota final)

$$NF = \frac{PFCC(0{,}5) + PCP(0{,}4) + PCA(0{,}1)}{(1{,}0)}$$

Los pesos son relativos a la naturaleza de las competencias Art. N°18,20,21 del Reglamento de Evaluación y aprendizaje.

Especificaciones sobre sustitutorio
Remitirse al reglamento de Evaluación del Aprendizaje (Resolución N°178-2010-CU-UNAMBA)

Abancay, 20 de Febrero del 2016

UNIVERSIDAD NACIONAL MICAELA BASTIDAS

ITEM DE EVALUACIÓN								
CAPACIDAD						ACTITUD		
CC	CONTENIDO CONCEPTUAL		CP	CONTENIDO PROCEDIMENTAL		CA	CONTENIDO ACTITUDINAL	
	ITEM	EVAL		ITEM	EVAL		ITEM	EVAL
I	A	3	I	G	3	I	A	2
II			II			II		
III			III					

N°	Código Estudiante	RESUMEN NOTAS					Promedio Final	NOTA FINAL	
		PCC	Sust.	PFCC	PCP	PCA		En Nros.	En letras
26	151153	08		08	07	18	09	09	Nueve
27	142187	10		10	15	14	12	12	Doce
28	151158	NSP		NSP	NSP	NSP	NSP	NSP	No Se Presentó

CAPACIDAD		ACTITUD
CONTENIDO CONCEPTUAL	CONTENIDO PROCEDIMENTAL	CONTENIDO ACTITUDINAL
A.-Pruebas Escritas	G.-Demostración de procesos	A.-Responde a las normas de convivencia

Observaciones

CONSOLIDADO	En Nros
Estudiantes aprobados	18
Estudiantes desaprobados	3
Estudiantes reprobados	4
Estudiantes NSP	3
Otros	
Total de Estudiantes matriculados	28

LEYENDA	
PI, PII, PIII	Promedio de Sub – contenido Conceptual, Procedimental, Actitudinal
PCC	Promedio de Contenido Conceptual
PCP	Promedio de Contenido Procedimental
PCA	Promedio de Contenido Actitudinal
PFCC	Promedio final de Contenido Conceptual
Sust.	Sustitutorio
CC, CP, CA	Nro. de Contenido Conceptual, Procedimental y Actitudinal

Fórmula del PCC	Promedio Aritmetico
Fórmula del PCP	Promedio Aritmetico
Fórmula del PCA	Promedio Aritmetico

Fórmula de la Nota Final (La fórmula del promedio final es la misma de la nota final)

$$NF = \frac{PCC}{PFCC(\ 0{,}5\ ) + PCP(\ 0{,}4\ ) + PCA(\ 0{,}1\ )}{(\ 1{,}0\ )}$$

Los pesos son relativos a la naturaleza de las competencias Art. N°18,20,21 del Reglamento de Evaluación y aprendizaje.

Especificaciones sobre sustitutorio
Remitirse al reglamento de Evaluación del Aprendizaje (Resolución N°178-2010-CU-UNAMBA)

Abancay, 20 de Febrero del 2016

UNIVERSIDAD NACIONAL MICAELA BASTIDAS

ANEXO C

Código Fuente del Compilador

Analizador Léxico

```
public class Lexico{
        public static String ExpresionRegularPalabra(){
            return @"[a-zA-Z]";
        }
        public static String ExpresionRegularNumero(){
            return @"\d{1}|\d{2}|\d{3}|\d{4}|\d{5}";
        }
        public string Analizador(char[] caracteres,string lenguaje, string codigoCP){
            string CodigoTraducido = "";
            string textoComillas = "";
            string textoCodigo = "";
            char comillas = '"';
            bool BanderaGraficos = false;
            string textoComentarioLinea = "";
            string textoComentarioParrafo = "";
            for (int i = 0; i < caracteres.Length - 1; i++){
                if (caracteres[i] == '"'){
                    textoComillas = textoComillas + caracteres[i];
                    i++;
                    while (caracteres[i] != '"'){
                        textoComillas = textoComillas + caracteres[i];
                        i++;
                        if (i == caracteres.Length){
                            break;
                        }
                    }
                    textoCodigo = textoCodigo + textoComillas + comillas;
                    textoComillas = "";
                }
                else if (caracteres[i] == '/'){
                    i++;
                    if (i == caracteres.Length){
                        textoCodigo = textoCodigo + "/";
                        break;
                    }else{
                        if (caracteres[i] == '/'){
                            textoCodigo = textoCodigo + caracteres[i - 1];
                            textoCodigo = textoCodigo + caracteres[i];
                            i++;
                            if (i == caracteres.Length){
                                break;
                            }else{
                                while (caracteres[i] != '\n'){
                                textoComentarioLinea = textoComentarioLinea + caracteres[i];
                                    i++;
                                    if (i == caracteres.Length){
                                        break;
                                    }
                                }
                            }
                            textoCodigo = textoCodigo + textoComentarioLinea;
                            textoCodigo = textoCodigo + "\n";
                            textoComentarioLinea = "";
                        }
                        else if (caracteres[i] == '*'){
                            textoCodigo = textoCodigo + caracteres[i - 1];
                            textoCodigo = textoCodigo + caracteres[i];
                            i++;
```

```
                        if (i == caracteres.Length){
                            break;
                        }else{
                            Boolean BanderaComentarioParrafo = true;
                            while (BanderaComentarioParrafo == true){
                                Boolean BanderaPrimaria = false;
                                Boolean BanderaSecundaria = false;
                                if (caracteres[i] == '*'){
                                    BanderaPrimaria = true;
                                }
                                if (caracteres[i + 1] == '/'){
                                    BanderaSecundaria = true;
                                }
                                if (BanderaPrimaria == true && BanderaSecundaria == true){
                                    BanderaComentarioParrafo = false;
                                }
                                i++;
                                if (i + 1 == caracteres.Length){
                                    textoComentarioParrafo = textoComentarioParrafo +
                                    caracteres[i];
                                    break;
                                }else{
                                    textoComentarioParrafo = textoComentarioParrafo +
                                    caracteres[i];
                                }
                            }
                        }
                        textoCodigo = textoCodigo + textoComentarioParrafo;
                        textoComentarioParrafo = "";
                    }else{
                        textoCodigo = textoCodigo + caracteres[i - 1];
                        i--;
                    }
                }
            }else{
                 textoCodigo = textoCodigo + caracteres[i];
                }
            }
        }
    }
}
```

Analizador sintáctico

```
public class Sintactico
    {
        public static string ExpresionRegularSentenciaEntero(){
            return @"^entero\s+[a-z](1,15)(\s+:\s+\d(0,32000))*;$";
        }
        public static string ExpresionRegularSentenciaDouble(){
            return @"^real\s+[a-z](1,15)(\s+:\s+\d(0,32000))*;$";
        }
        public static string ExpresionRegularSentenciaString(){
            return @"^cadena\s+[a-z](1,15)(\s+:\s+[a-z](1,15)')*;$";
        }
        public static string ExpresionRegularSentenciaBoolean(){
            return @"^boleano\s+[a-z](1,15)(\s+:\s+(true|false))*;$";
        }
        public static string ExpresionRegularSentenciaComentario(){
```

```
            return @"/*.*/$";
        }
        public static string ExpresionRegularSentenciaAsignacion(){
            return @"[a-z]\s+:\s[a-z]|(\w)*\s\+\s(\w)*|\d(0,32000)*\s;$";
        }
        public static string ExpresionRegularInicioDeAmbito(){
            return @"^Inicio$";
        }
        public static string ExpresionRegularFinDeAmbito(){
            return @"^Fin$";
        }
        public static string ExpresionRegularComienzoDeIf(){
            return @"<<Si\s\(\s+\w+\s(<|>|<:|>:|::|!:)\s\w+\s\)\s\Entonces$";
        }
        public static string ExpresionRegularComienzoDeElse(){
            return @"<<Sino\s*\FinSi$";
        }
        public static string ExpresionRegularComienzoDeSwitch(){
            return @"#Segun\s\(\s\w+\s(<|>|<:|>:|::|!:)\s\w+\s\)\s\Hacer$";
        }
        public static string ExpresionRegularComienzoDeCase(){
            return @"caso\s\(\s(\w+|\d+)\s\)\s{$";
        }
        public static string ExpresionRegularComienzoDeBreakCase(){
            return @"romper\s;$";
        }
        public static string ExpresionRegularComienzoDeWhile(){
            return @"#Mientras\s\(\s\w+\s(<|>|<:|>:|::|!:)\s\w+\s\)\sHacer$";
        }
        public static string ExpresionRegularMostrarPorPantalla(){
            return @"#Escribir\s\(\s(\w*)|'\w*'\)\s;$" ;
        }

    }
```

Analizador Semántico

```
public class AnalizadorSemantico{
        private Form hijo;
        public Form Hijo{
            get { return hijo; }
            set { hijo = value; }
        }
        public void SemanticaSentenciaEntero(string sentencia,int i){
            string[] separanum;
            separanum = sentencia.Split(' ');
            try{
                int num;
                num = int.Parse(separanum[3]);
              }catch (FormatException){
              ((MenuPrincipal)Hijo).tabla_errorres.AgregarListaTablaErroresPseudocodigo
              (0, i);
            }catch (IndexOutOfRangeException)
            {
              ((MenuPrincipal)Hijo).tabla_errorres.AgregarListaTablaErroresPseudocodigo
              (10, i);
            }
        }
        public void SemanticaSentenciaDouble(string sentencia,int i){
```

```
        string[] separanum;
        separanum = sentencia.Split(' ');
        try{
            double num;
            num = double.Parse(separanum[3]);
        }
        catch (FormatException)
        {
          ((MenuPrincipal)Hijo).tabla_errorres.AgregarListaTablaErroresPseudocodigo
          (0, i);
        }
        catch (IndexOutOfRangeException){
          ((MenuPrincipal)Hijo).tabla_errorres.AgregarListaTablaErroresPseudocodigo
          (10, i);
        }
    }
    public void SemanticaSentenciaBooleano(string sentencia,int i){
        string[] separavar;
        separavar = sentencia.Split(' ');
        try{
            bool var;
            var = bool.Parse(separavar[3]);
        }catch (FormatException){
          ((MenuPrincipal)Hijo).tabla_errorres.AgregarListaTablaErroresPseudocodigo
          (0, i);
        }
    }
    public void SemanticaSentenciaAsignacion(string sentencia){
        string tpv1 = "";
        string tpv2 = "";
        string tpv3 = "";
        string[] separavar;
        separavar = sentencia.Split(' ');
        if (Regex.IsMatch(sentencia,Semantico.ExpresionRegularAsignacionTipoSuma()))
        {
            foreach (var token in ((MenuPrincipal)Hijo).tabla_simbolos.
            TablaSimbolosListaClase()){

                if (token.Simbolo == separavar[0]){tpv1 = token.TipoValor;}
                if (token.Simbolo == separavar[2]){tpv2 = token.TipoValor;}
                if (token.Simbolo == separavar[4]){tpv3 = token.TipoValor;}
            }
            if (tpv1 == tpv2 && tpv2 == tpv3 && tpv1 != ""){
                //MessageBox.Show("el tipo de las variables son el mismo");
            }
        }
        if(Regex.IsMatch(sentencia,Semantico.ExpresionRegularAsignacionTipoResta()))
        {
          foreach (var token in ((MenuPrincipal)Hijo).tabla_simbolos.
          TablaSimbolosListaClase()){
                if (token.Simbolo == separavar[0]){
                    tpv1 = token.TipoValor;
                }
                if (token.Simbolo == separavar[2]){
                    tpv2 = token.TipoValor;
                }
                if (token.Simbolo == separavar[4]){
                    tpv3 = token.TipoValor;
                }

            }
            if (tpv1 == tpv2 && tpv2 == tpv3 && tpv1 != ""){
```

```
                    //MessageBox.Show("el tipo de las variables son el mismo");
                }
                }
                if(Regex.IsMatch(sentencia,Semantico.ExpresionRegularAsignacionTipoDivi
                sion())){
                 foreach (var token in ((MenuPrincipal)Hijo).tabla_simbolos.
                 TablaSimbolosListaClase()){
                    if (token.Simbolo == separavar[0]){
                        tpv1 = token.TipoValor;
                    }
                    if (token.Simbolo == separavar[2]){
                        tpv2 = token.TipoValor;
                    }
                    if (token.Simbolo == separavar[4]){
                        tpv3 = token.TipoValor;
                    }

                 }//-- fin del foreach
                 if (tpv1 == tpv2 && tpv2 == tpv3 && tpv1 != ""){
                    //MessageBox.Show("el tipo de las variables son el mismo");
                 }
                 }
                 if(Regex.IsMatch(sentencia,Semantico.ExpresionRegularAsignacionTipoMul
                 tiplicasion())){
                 foreach (var token in ((MenuPrincipal)Hijo).tabla_simbolos.
                 TablaSimbolosListaClase()){
                    if (token.Simbolo == separavar[0]){
                        tpv1 = token.TipoValor;
                    }
                    if (token.Simbolo == separavar[2]){
                        tpv2 = token.TipoValor;
                    }
                    if (token.Simbolo == separavar[4]){
                        tpv3 = token.TipoValor;
                    }
                 }
                 if (tpv1 == tpv2 && tpv2 == tpv3 && tpv1 != ""){
                    //MessageBox.Show("el tipo de las variables son el mismo");
                 }
             }
         }
}
public class Semantico
    {
        /// <summary>
        /// Expresion regular para asignacion del tipo monto : numero1 + numero2;
        /// </summary>
        /// <returns>Retorna la expresion regular del tipo suma</returns>
        public static String ExpresionRegularAsignacionTipoSuma()
        {
            return @"[a-z]\s+:\s(\w)*\s\+\s(\w)*\s;$";
        }
        /// <summary>
        /// Expresion regular para asignacion del tipo monto : numero1 - numero2;
        /// </summary>
        /// <returns>Retorna la expresion regular del tipo resta</returns>
        public static String ExpresionRegularAsignacionTipoResta()
        {
            return @"[a-z]\s+:\s(\w)*\s\-\s(\w)*\s;$";
        }
        /// <summary>
        /// Expresion regular para asignacion del tipo monto : numero1 / numero2;
```

```
        /// </summary>
        /// <returns>Retorna la expresion regular del tipo division</returns>
        public static String ExpresionRegularAsignacionTipoDivision()
        {
            return @"[a-z]\s+:\s(\w)*\s\/\s(\w)*\s;$";
        }
        public static String ExpresionRegularAsignacionTipoMultiplicasion()
        {
            return @"[a-z]\s+:\s(\w)*\s\*\s(\w)*\s;$";
        }
    }
```

Tabla de Símbolos

```
    public class TablaSimbolos{
        public List<ClaseTablaSimbolos> ListaTablaSimbolos = new
        List<ClaseTablaSimbolos>();
        TablaErrores tablaErrorres = new TablaErrores();
        public TablaSimbolos(){}
        public void LimpiarListaTablaSimbolos(){
          ListaTablaSimbolos.Clear();
        }
        public void IniciaListaTablaSimbolos()
        {
           ClaseTablaSimbolos objts = new ClaseTablaSimbolos("/*", "", -0, -0, -0, 0,
           "Comentario", "Inicio de un comentario de mas de una linea", "");
           ListaTablaSimbolos.Add(objts);
           ClaseTablaSimbolos objts1 = new ClaseTablaSimbolos("*/", "", -0, -0, -0, 1,
           "Comentario", "Final de un comentario de mas de una linea", "");
           ListaTablaSimbolos.Add(objts1);
           ClaseTablaSimbolos objts2 = new ClaseTablaSimbolos("//", "", -0, -0, -0, 2,
           "Comentario", "Inicio de un comentario de una linea", "");
           ListaTablaSimbolos.Add(objts2);
           ClaseTablaSimbolos objts6 = new ClaseTablaSimbolos("entero", "", -0, -0, -
           0, 6, "Palabra reservada", "Numero entero", "");
           ListaTablaSimbolos.Add(objts6);
           ClaseTablaSimbolos objts7 = new ClaseTablaSimbolos("real", "", -0, -0, -0,
           7, "Palabra reservada", "Numero real", "");
           ListaTablaSimbolos.Add(objts7);
           ClaseTablaSimbolos objts8 = new ClaseTablaSimbolos("double", "", -0, -0, -
           0, 8, "Palabra reservada", "Numero con decimales", "");
           ListaTablaSimbolos.Add(objts8);
           ClaseTablaSimbolos objts9 = new ClaseTablaSimbolos("cadena", "", -0, -0, -
           0, 9, "Palabra reservada", "Cadena de caracteres", "");
           ListaTablaSimbolos.Add(objts9);
           ClaseTablaSimbolos objts10 = new ClaseTablaSimbolos("booleano", "", -0, -0,
           -0, 10, "Palabra reservada", "Booleano verdadero o falso", "");
           ListaTablaSimbolos.Add(objts10);
           ClaseTablaSimbolos objts11 = new ClaseTablaSimbolos("caracter", "", -0, -0,
           -0, 11, "Palabra reservada", "Caracteres", "");
           ListaTablaSimbolos.Add(objts11);
           ClaseTablaSimbolos objts12 = new ClaseTablaSimbolos("=", "", -0, -0, -0,
           12, "Asignacion", "Simbolo de asignacion", "");
           ListaTablaSimbolos.Add(objts12);
           ClaseTablaSimbolos objts13 = new ClaseTablaSimbolos(";", "", -0, -0, -0,
           13, "Posicionador", "Final de linea", "");
           ListaTablaSimbolos.Add(objts13);
           ClaseTablaSimbolos objts18 = new ClaseTablaSimbolos("+", "", -0, -0, -0,
           18, "Operador", "Suma", "");
           ListaTablaSimbolos.Add(objts18);
```

```
ClaseTablaSimbolos objts19 = new ClaseTablaSimbolos("+", "", -0, -0, -0,
19, "Concatenador", "Concatenador de elementos","");
ListaTablaSimbolos.Add(objts19);
ClaseTablaSimbolos objts20 = new ClaseTablaSimbolos("-", "", -0, -0, -0,
20, "Operador", "Resta", "");
ListaTablaSimbolos.Add(objts20);
ClaseTablaSimbolos objts21 = new ClaseTablaSimbolos("*", "", -0, -0, -0,
21, "Operador", "Multiplicacion", "");
ListaTablaSimbolos.Add(objts21);
ClaseTablaSimbolos objts22 = new ClaseTablaSimbolos("/", "", -0, -0, -0,
22, "Operador", "Division", "");
ListaTablaSimbolos.Add(objts22);
ClaseTablaSimbolos objts23 = new ClaseTablaSimbolos("!", "", -0, -0, -0,
23, "Signo relacionador", "Negacion", "");
ListaTablaSimbolos.Add(objts23);
ClaseTablaSimbolos objts24 = new ClaseTablaSimbolos(">", "", -0, -0, -0,
24, "Signo comparador", "Mayor que", "");
ListaTablaSimbolos.Add(objts24);
ClaseTablaSimbolos objts25 = new ClaseTablaSimbolos("<", "", -0, -0, -0,
25, "Signo comparador", "Menor que", "");
ListaTablaSimbolos.Add(objts25);
ClaseTablaSimbolos objts26 = new ClaseTablaSimbolos(">=", "", -0, -0, -0,
26, "Signo comparador", "Mayor o igual que", "");
ListaTablaSimbolos.Add(objts26);
ClaseTablaSimbolos objts27 = new ClaseTablaSimbolos("<=", "", -0, -0, -0,
27, "Signo comparador", "Menor o igual que", "");
ListaTablaSimbolos.Add(objts27);
ClaseTablaSimbolos objts28 = new ClaseTablaSimbolos("&", "", -0, -0, -0,
28, "Signo relacionador", "Expresion y", "");
ListaTablaSimbolos.Add(objts28);
ClaseTablaSimbolos objts29 = new ClaseTablaSimbolos("||", "", -0, -0, -0,
29, "Signo relacionador", "Expresion o", "");
ListaTablaSimbolos.Add(objts29);
ClaseTablaSimbolos objts30 = new ClaseTablaSimbolos("==", "", -0, -0, -0,
30, "Signo Comparador", "Expresion igual", "");
ListaTablaSimbolos.Add(objts30);
ClaseTablaSimbolos objts31 = new ClaseTablaSimbolos("!=", "", -0, -0, -0,
31, "Signo comparador", "Expresion distinto", "");
ListaTablaSimbolos.Add(objts31);
ClaseTablaSimbolos objts32 = new ClaseTablaSimbolos("Proceso", "", -0, -0,
-0, 32, "Palabra reservada", "Si tal condicion se cumple", "");
ListaTablaSimbolos.Add(objts32);
ClaseTablaSimbolos objts33 = new ClaseTablaSimbolos("Principal", "", -0, -
0, -0, 33, "Palabra reservada", "Inicio de condicional", "");
ListaTablaSimbolos.Add(objts33);
ClaseTablaSimbolos objts34 = new ClaseTablaSimbolos("FinProceso", "", -0, -0,
-0, 34, "Palabra reservada", "Si no se cumple la condicion", "");
ListaTablaSimbolos.Add(objts34);
ClaseTablaSimbolos objts35 = new ClaseTablaSimbolos("Escribir", "", -0, -0,
-0, 35, "palabra reservada", "opciones (casos)", "");
ListaTablaSimbolos.Add(objts35);
ClaseTablaSimbolos objts36 = new ClaseTablaSimbolos("Leer", "", -0, -0, -0,
36, "palabra reservada", "caso", "");
ListaTablaSimbolos.Add(objts36);
ClaseTablaSimbolos objts37 = new ClaseTablaSimbolos("Si", "", -0, -0, -0,
37, "palabra reservada", "final del caso", "");
ListaTablaSimbolos.Add(objts37);
ClaseTablaSimbolos objts38 = new ClaseTablaSimbolos("Entonces", "", -0, -0,
-0, 38, "palabra reservada", "mientras", "");
ListaTablaSimbolos.Add(objts38);
ClaseTablaSimbolos objts39 = new ClaseTablaSimbolos("Sino", "", -0, -0, -0,
39, "palabra reservada", "for", "");
```

```
            ListaTablaSimbolos.Add(objts39);
            ClaseTablaSimbolos objts40 = new ClaseTablaSimbolos("FinSi", "", -0, -0, -
            0, 40, "palabra reservada", "foreach", "");
            ListaTablaSimbolos.Add(objts40);
            ClaseTablaSimbolos objts41 = new ClaseTablaSimbolos("Segun", "", -0, -0, -0,
            41, "palabra reservada", "expresion para determinar un metodo", "");
            ListaTablaSimbolos.Add(objts41);
            ClaseTablaSimbolos objts42 = new ClaseTablaSimbolos("Hacer", "", -0, -0, -
            0, 42, "palabra reservada", "intenta realizar el codigo contenido", "");
            ListaTablaSimbolos.Add(objts42);
            ClaseTablaSimbolos objts43 = new ClaseTablaSimbolos("Caso", "", -0, -0, -0,
            43, "palabra reservada", "muestra una accion a seguir en caso de error",
            "");
            ListaTablaSimbolos.Add(objts43);
            ClaseTablaSimbolos objts44 = new ClaseTablaSimbolos("Parar", "", -0, -0, -
            0, 44, "Palabra reservada", "Muestra en consola", "");
            ListaTablaSimbolos.Add(objts44);
            ClaseTablaSimbolos objts45 = new ClaseTablaSimbolos("FinSegun", "", -0, -0,
            -0, 45, "Palabra reservada", "Pide en consola", "");
            ListaTablaSimbolos.Add(objts45);
        }
```

Manejador de Errores

```
namespace ManejadorDeErrores{
    public class TablaErrores{
       private List<ClaseTablaErrores> ListaTablaErrores = new
       List<ClaseTablaErrores>();
       private List<ClaseTablaErrores> ListaTablaErroresPseudocodigo = new
       List<ClaseTablaErrores>();
       public List<ClaseTablaErrores> TablaErroresLista{
             get { return ListaTablaErrores; }
             set { ListaTablaErrores = value; }
       }
       public void LimpiarListaTablaErrores(){
           ListaTablaErrores.Clear();
           ListaTablaErroresPseudocodigo.Clear();
       }
       public void IniciaListaTablaErrores(){
           ListaTablaErrores.Clear();
           ClaseTablaErrores objte = new ClaseTablaErrores(0, "valor incorrecto",
           "escriba el valor aceptado por el tipo de variable", "valor diferente al
           aceptado por el tipo");
           ListaTablaErrores.Add(objte);
           ClaseTablaErrores objte1 = new ClaseTablaErrores(1, "se espera un valor",
           "escriba un valor para la variable ", "se espera un valor despues de...");
           ListaTablaErrores.Add(objte1);
           ClaseTablaErrores objte2 = new ClaseTablaErrores(2, "error al abrir",
           "revise la extencion del archivo o  la direccion del mismo", "error al
           abrir el archivo");
           ListaTablaErrores.Add(objte2);
           ClaseTablaErrores objte3 = new ClaseTablaErrores(3, "error aritmetico",
           "revise la operación que esta realizando", "excepciones producidas durante
           operaciones aritméticas");
           ListaTablaErrores.Add(objte3);
           ClaseTablaErrores objte4 = new ClaseTablaErrores(4, "error dividir por cero
           ", "escoja otro numero que no sea el 0 para dividir", "posible
           incongruencia en divicion,o en cualquier operación");
           ListaTablaErrores.Add(objte4);
```

```
ClaseTablaErrores objte5 = new ClaseTablaErrores(5, "error de conversion de
tipo", "verifique que los tipos de las variables sea el mismo ", "Se
produce cuando tiene lugar un error en tiempo de ejecución en una
conversión explícita de un tipo base a una interfaz o a un tipo
derivado.");
ListaTablaErrores.Add(objte5);
ClaseTablaErrores objte6 = new ClaseTablaErrores(6, "error referencia
nula", "revise que esta dando un valor ala variable", "Se produce al
intentar hacer referencia a un objeto cuyo valor es null.");
ListaTablaErrores.Add(objte6);
ClaseTablaErrores objte7 = new ClaseTablaErrores(7, "error de
desbordamiento", "asegurese del tamaño del resultado ", "Se produce cuando
una operación aritmética en un contexto produce un desbordamiento.");
ListaTablaErrores.Add(objte7);
ClaseTablaErrores objte8 = new ClaseTablaErrores(8, "error de Ambito",
"asegurese de que las llaves '{' tengan su contraparte '}' ", "Se produce
cuando hay alguna llave sin cerrar, ambito incompleto");
ListaTablaErrores.Add(objte8);
ClaseTablaErrores objte9 = new ClaseTablaErrores(9, "sintaxis desconocida",
"asegurese de que la sintaxis sea correcta ", "Se produce cuando se
desconoce la sintaxis de la sentencia");
ListaTablaErrores.Add(objte9);
ClaseTablaErrores objte10 = new ClaseTablaErrores(10, "sintaxis erronea",
"asegurese de que la sintaxis sea correcta, verifique espacios ", "Se
produce cuando la sintaxis de la sentencia contiene algun error");
ListaTablaErrores.Add(objte10);
ClaseTablaErrores objte11 = new ClaseTablaErrores(11, "warning", "asegurese
de que las variables no esten repetidas ", "Se produce cuando mas de una
variable estan inicializadas con el mismo nombre");
ListaTablaErrores.Add(objte11);
ClaseTablaErrores objte12 = new ClaseTablaErrores(12, "warning", "Se ha
alcanzado el final del archivo. Posible algoritmo incompleto", "");
ListaTablaErrores.Add(objte12);
ClaseTablaErrores objte13 = new ClaseTablaErrores(13, "warning", "Este
número es demasiado grande", "");
ListaTablaErrores.Add(objte13);
ClaseTablaErrores objte14 = new ClaseTablaErrores(14, "warning", "El máximo
número de cifras decimales ha sido sobrepasado", "");
ListaTablaErrores.Add(objte14);
ClaseTablaErrores objte15 = new ClaseTablaErrores(15, "warning", "La cadena
es demasiado larga", "");//F
ListaTablaErrores.Add(objte15);
ClaseTablaErrores objte16 = new ClaseTablaErrores(16, "warning", "Falta la
comilla de cierre de cadena", "");//F
ListaTablaErrores.Add(objte16);
ClaseTablaErrores objte17 = new ClaseTablaErrores(17, "warning", "Falta la
comilla de cierre de caracter", "");//F
ListaTablaErrores.Add(objte17);
ClaseTablaErrores objte20 = new ClaseTablaErrores(20, "warning", "Se
esperaba 'inicio'", "");
ListaTablaErrores.Add(objte20);
ClaseTablaErrores objte21 = new ClaseTablaErrores(21, "warning", "Se
esperaba un identificador", "");
ListaTablaErrores.Add(objte21);
ClaseTablaErrores objte22 = new ClaseTablaErrores(22, "warning", "Se
esperaba )", "");
ListaTablaErrores.Add(objte22);
ClaseTablaErrores objte23 = new ClaseTablaErrores(23, "warning", "Error
sintáctico, se esperaba '('", "");
ListaTablaErrores.Add(objte23);
ClaseTablaErrores objte24 = new ClaseTablaErrores(24, "warning", "Se
esperaba }", "");
```

```
            ListaTablaErrores.Add(objte24);
        }
        public void AgregarListaTablaErroresPseudocodigo(int idError){
            foreach (var error in ListaTablaErrores){
                if (error.IdError == idError){
                    ClaseTablaErrores objError = new ClaseTablaErrores();
                    objError.Descripcion = error.Descripcion;
                    objError.SolucionError = error.SolucionError;
                    objError.IdError = error.IdError;
                    ListaTablaErroresPseudocodigo.Add(objError);
                }
            }
        }
    }
}
```

Código Compilación en Ejecutable Exe

```
public ArrayList Compilar(string CodigoFuente,string Direccion,string
NombreArchivo,Boolean Ejecutable){
             ArrayList ListaErrores = new ArrayList();
             CSharpCodeProvider codeProvider = new CSharpCodeProvider();
             ICodeCompiler icc = codeProvider.CreateCompiler();
             string Output = Explorador.DireccionMisDocumentos()+
             "\\HITO\\Compilados\\"+NombreArchivo  + ".exe";
             System.CodeDom.Compiler.CompilerParameters parameters = new
             CompilerParameters();
             parameters.GenerateExecutable = true;
             parameters.OutputAssembly = Output;
             parameters.ReferencedAssemblies.Add("System.dll");
             parameters.ReferencedAssemblies.Add("System.Windows.Forms.dll");
             parameters.ReferencedAssemblies.Add("System.Drawing.dll");
             parameters.ReferencedAssemblies.Add("System.Drawing.Design.dll");
             CompilerResults results = icc.CompileAssemblyFromSource(parameters,
             CodigoFuente);
             if (results.Errors.Count > 0){
                 foreach (CompilerError CompErr in results.Errors){
                     ClaseErrores objErrores = new
                     ClaseErrores(Automator.Control.Generador.GeneraID(),
                     CompErr.ErrorText, CompErr.Line, true);
                     ListaErrores.Add(objErrores);
                  }
             }else{
                 if (Ejecutable == true){
                     Process.Start(Output);
                 }
             }
             return ListaErrores;
         }
```

ANEXO D

Pruebas unitarias del compilador de pseudocódigo

Las pruebas unitarias se realizaron para los diferentes componentes del compilador de pseudocódigo:

- ✓ Componente del analizador léxico
- ✓ Componente del analizado sintáctico
- ✓ Componente del analizador semántico
- ✓ Componente de la tabla se símbolos
- ✓ Componente de la tabla de errores
- ✓ Componente traductor
- ✓ Componente del generador binario
- ✓ Componente automator
- ✓ Componente editor UI

Para lo cual se utilizó la herramienta Unit Test con la que cuenta Visual Studio, también se utilizó el plugin DevExpressCodeRush que permite ejecutar los casos de prueba en archivos DLL separados al mismo tiempo para que pueda volver a la codificación antes. El desarrollo basado en pruebas de CodeRush permite crear nuevos casos de prueba como se muestra en la siguiente figura:

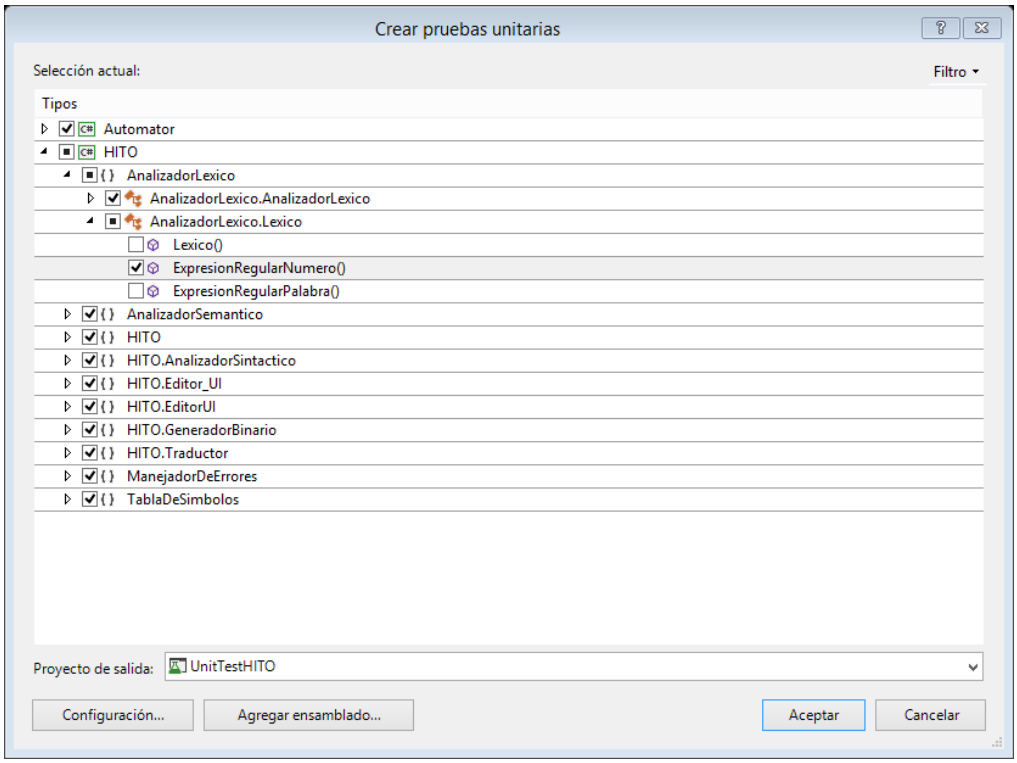


Figura N° 30: Creación de pruebas unitarias en Visual Studio

Fuente: Elaboración propia

En la Figura N° 31 se muestra los diferentes componentes del compilador de pseudocódigo al cual se realizó las pruebas unitarias. Obteniendo los siguientes resultados:

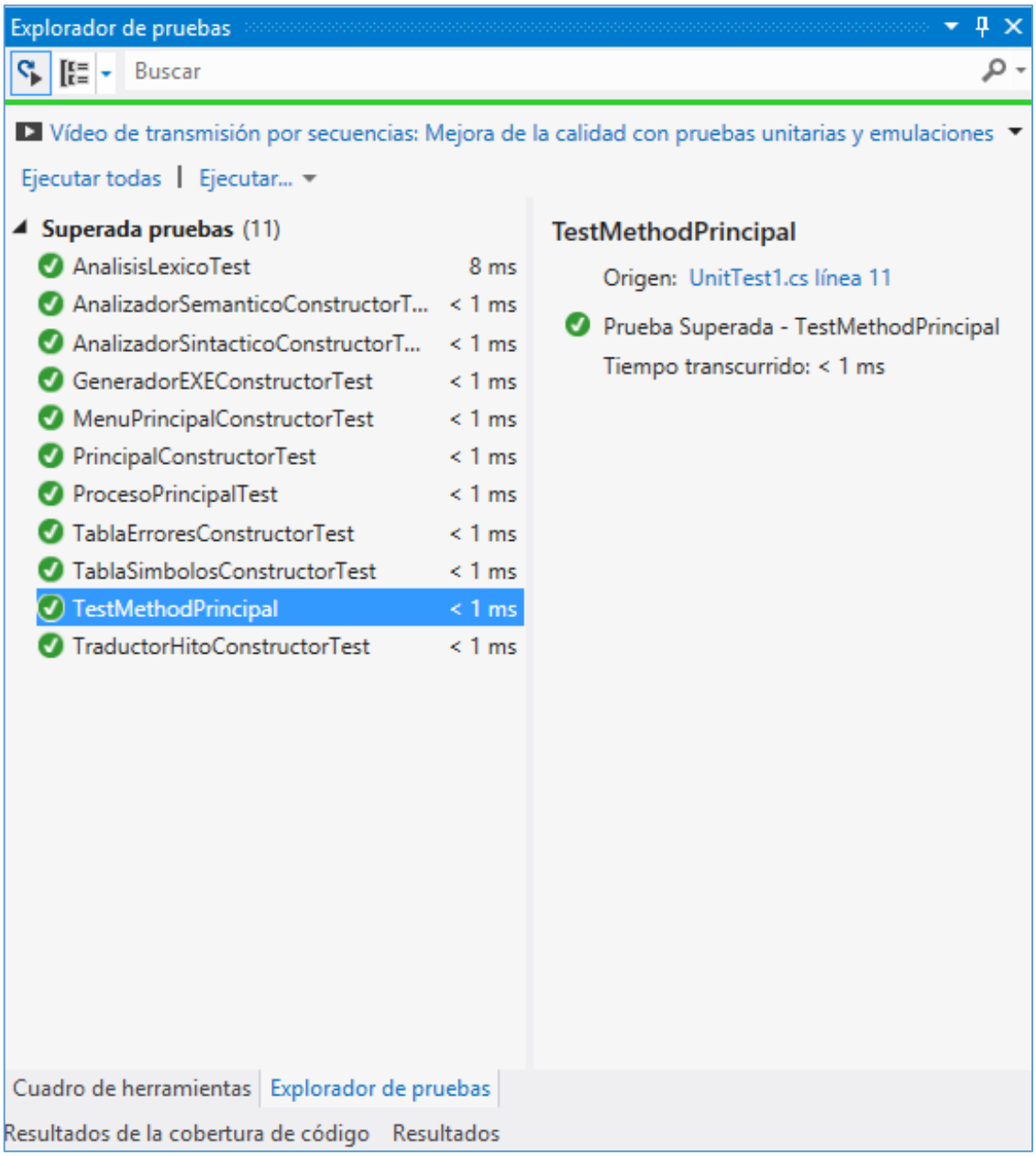


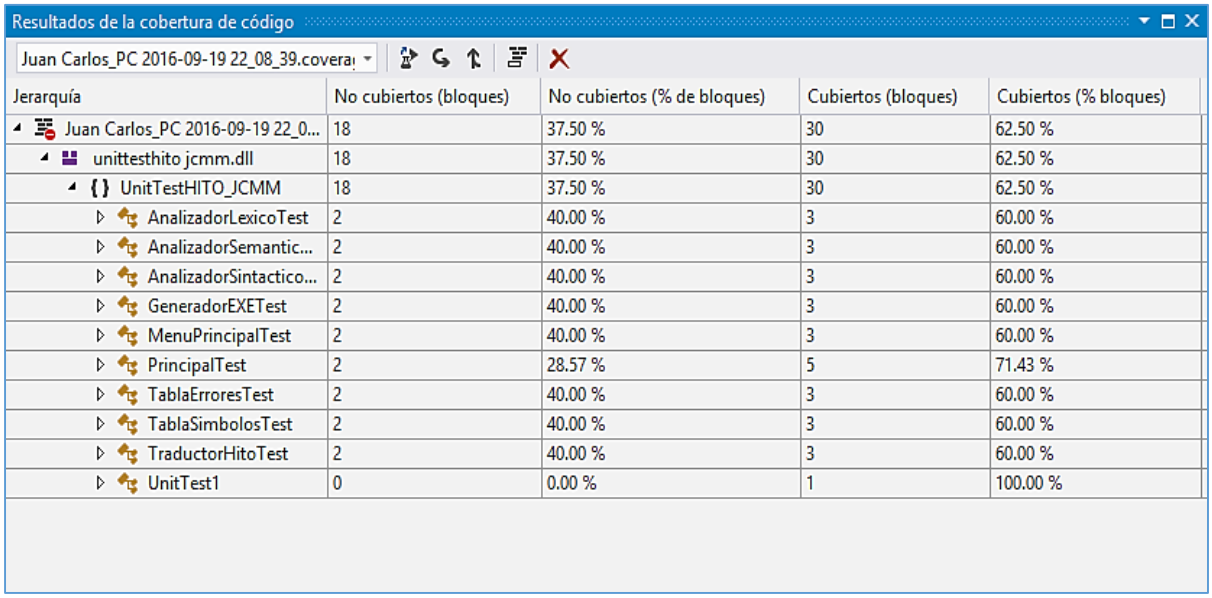
Figura N° 31 : Explorador de pruebas unitarias

Fuente: Elaboración propia

Cobertura de código

Para determinar qué proporción de código del proyecto se está probando realmente mediante pruebas codificadas como pruebas unitarias, se utilizó la característica de cobertura de código de Visual Studio. El cual nos permitió restringir con eficacia los errores y cubrir una proporción considerable del código.

La cobertura de código es una opción con la cuenta Visual Studio al ejecutar métodos de prueba mediante el Explorador de pruebas. La tabla de salida muestra el porcentaje de código que se ejecuta en cada ensamblado, clase y método.



Jerarquía	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% bloques)
Juan Carlos_PC 2016-09-19 22_0...	18	37.50 %	30	62.50 %
unittesthito jcmm.dll	18	37.50 %	30	62.50 %
UnitTestHITO_JCMM	18	37.50 %	30	62.50 %
AnalizadorLexicoTest	2	40.00 %	3	60.00 %
AnalizadorSemantic...	2	40.00 %	3	60.00 %
AnalizadorSintactico...	2	40.00 %	3	60.00 %
GeneradorEXETest	2	40.00 %	3	60.00 %
MenuPrincipalTest	2	40.00 %	3	60.00 %
PrincipalTest	2	28.57 %	5	71.43 %
TablaErroresTest	2	40.00 %	3	60.00 %
TablaSimbolosTest	2	40.00 %	3	60.00 %
TraductorHitoTest	2	40.00 %	3	60.00 %
UnitTest1	0	0.00 %	1	100.00 %

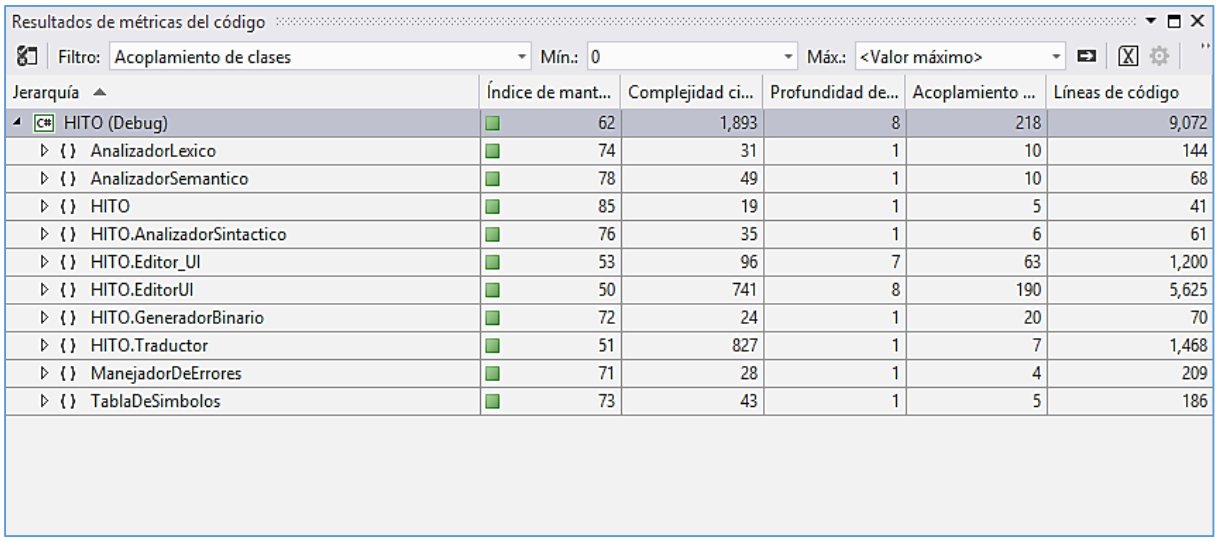
Figura N° 32 : Resultados de la cobertura de código

Fuente: Elaboración propia

Pruebas de sistema del compilador de pseudocódigo

Valores de métrica de código

Para poder tener una mejor visión del código fuente del compilador de pseudocódigo, se realizó el cálculo de valores de métricas de código. El cual nos permitió entender qué tipos y métodos se deben rehacer o probar más a fondo. Identificar los riesgos potenciales, entender el estado actual del proyecto y seguir el progreso durante el desarrollo del compilador de pseudocódigo. Para ello se utilizó la herramienta de cálculo de métricas con el que cuenta Visual Studio.


Resultados de métricas del código

Filtro: Acoplamiento de clases | Mín.: 0 | Máx.: <Valor máximo>

Jerarquía	Índice de mant...	Complejidad ci...	Profundidad de...	Acoplamiento ...	Líneas de código
HITO (Debug)	62	1,893	8	218	9,072
AnalizadorLexico	74	31	1	10	144
AnalizadorSemantico	78	49	1	10	68
HITO	85	19	1	5	41
HITO.AnalizadorSintactico	76	35	1	6	61
HITO.Editor_UI	53	96	7	63	1,200
HITO.EditorUI	50	741	8	190	5,625
HITO.GeneradorBinario	72	24	1	20	70
HITO.Traductor	51	827	1	7	1,468
ManejadorDeErrores	71	28	1	4	209
TablaDeSimbolos	73	43	1	5	186

Figura N° 33: Valores de métricas de código

Fuente: Elaboración propia

Análisis de rendimiento del compilador de pseudocódigo

Se utilizó la herramienta de generación de perfiles de Visual Studio para analizar problemas de rendimiento del compilador de pseudocódigo. Este procedimiento obtiene muestras de las funciones que realizan la mayor parte del trabajo por parte del usuario. Se recopila a intervalos especificados información sobre las funciones que se ejecutan en el compilador de pseudocódigo.

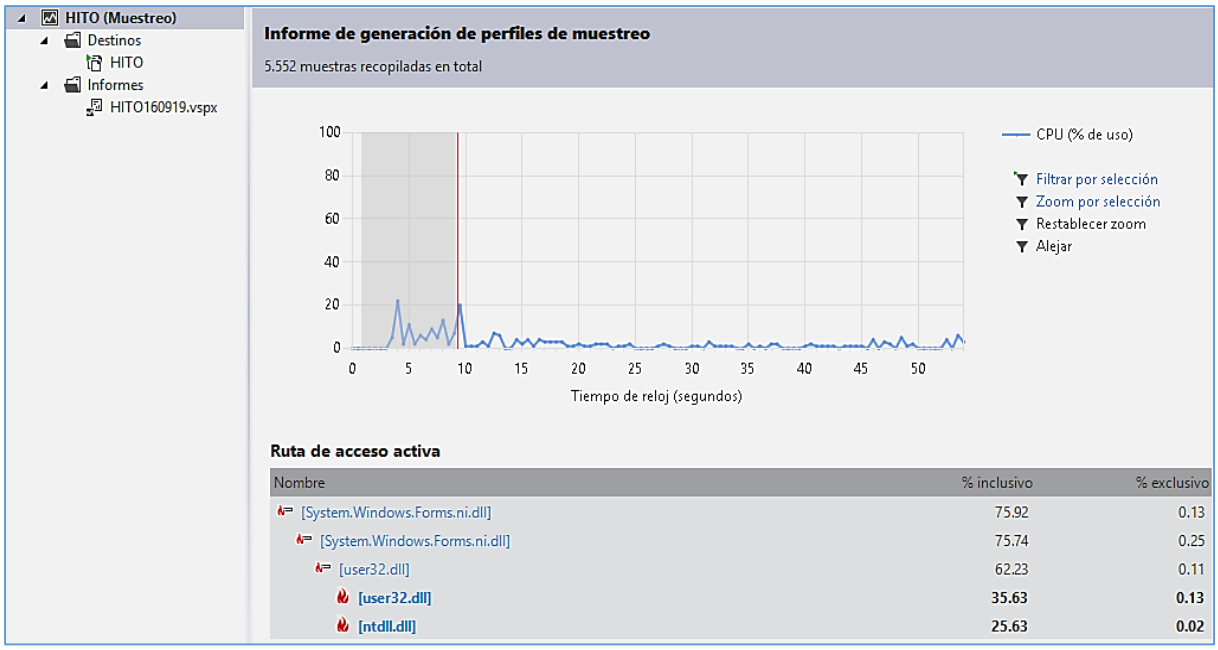


Figura N° 34: Análisis de rendimiento

Fuente: Elaboración propia

Muestras inclusivas: indica cuánto trabajo realizaron la función y las funciones llamadas por ella.

Muestras exclusivas: indica cuánto trabajo realizó el código de la función, sin incluir el trabajo de las funciones llamadas por dicha función.

Funcionalidades del compilador de pseudocódigo

Para poder mostrar las diferentes funcionalidades con las que cuenta el compilador de pseudocódigo, teniendo como base el historial de usuarios y las diferentes versiones del compilador de pseudocódigo, se obtiene la siguiente tabla:

Tabla N° 10 : Funcionalidades del compilador

	Versiones del compilador de pseudocódigo						
HU	**Versión 1.0**	**Versión 1.1**	**Versión 1.2**	**Versión 1.3**	**Versión 1.4**	**Versión 1.5**	**Versión 2.0**
HU1		x	x	x	x	x	x
HU2	X	x	x	x	x	x	x
HU3	X	x	x	x	x	x	x
HU4			x	x	x	x	x
HU5				x	x	x	x
HU6		x	x	x	x	x	x
HU7	X	x	x	x	x	x	x
HU8			x	x	x	x	x
HU9		x	x	x	x	x	x

HU10			x	x	x	x	x
HU11			x	x	x	x	x
HU12	X	x	x	x	x	x	x
HU13	X	x	x	x	x	x	x
HU14		x	x	x	x	x	x
HU15	X	x	x	x	x	x	x
HU16				x	x	x	x
HU17	X					x	x
HU18	X	x	x	x	x	x	x

Fuente: Elaboración propia

Los "x" significan que se cumplió con el desarrollo del historial de usuarios en cada una de las versiones del compilador de pseudocódigo.

Donde:

HU1. Al cerrar la aplicación: Si el pseudocódigo en el que se está trabajando no se ha guardado, preguntar si se desean guardar los cambios. Se puede realizar desde alguna opción de la barra de menú.

HU2. Al abrir la aplicación: Aparece la ventana principal con la barra de menú, que permitirá la creación y edición de pseudocódigo.

HU3. Archivo: Se deben poder crear, abrir y editar archivos de pseudocódigo.

HU4. Definición de archivos de Pseudocódigo: Archivos propios del compilador con extensión (.hito) y archivos planos de texto, que contienen un código fuente comprensible para la herramienta, la cual estará en la capacidad de realizar sobre estos análisis léxico, sintáctico y semántico, y ofrecerá la funcionalidad de compilación.

HU5. Archivos .hito: Los archivos con extensión hito (.hito); permitirán abrir automáticamente con el compilador y tendrán su propio icono.

HU6. Barra de menú de comandos y operadores y funciones: Permite agregar fácilmente códigos predefinidos para una rápita edición.

HU7. Vista de creación archivo de Pseudocódigo: Cuando se crea un archivo de pseudocódigo, la aplicación debe mostrar un editor del código, y debe proveer la estructura básica de un código fuente, realizando la coloración del texto.

HU8. Vista de edición de archivo de Pseudocódigo: Para editar un archivo de Pseudocódigo se debe mostrar el editor de código, junto con un grupo de herramientas, definidas a libre disposición del desarrollo, que promuevan la usabilidad de la herramienta.

HU9. Editar fuente de letra: La fuente se puede editar con las opciones que se encuentran en una barra ubicada en la parte superior de la herramienta en la cual se puede subir o bajar el tamaño de la fuente, cambiar el tipo de fuente.

HU10. Guardar archivos: Para guardar los archivos se puede realizar por medio del icono en la barra de herramientas o desde la opción de la barra de menú, se puede guardar los cambios en el mismo documento o realizar una copia del mismo.

HU11. Imprimir archivo: Se debe proveer la funcionalidad de impresión de archivos de código fuente así como de vista previa de impresión.

HU12. Verificar Sintaxis: Se puede realizar desde la barra de herramientas y desde la opción de la barra menú, haciendo click en el icono. En esta se realiza la estructura del programa a ejecutar y se muestran posibles errores sintácticos y semánticos que posea el código fuente escrito.

HU13. Ejecutar: Se puede realizar desde la barra de herramientas y desde la opción de la barra menú. En esta se realizará primero la compilación del archivo fuente, para después ser ejecutado el programa en caso que no haya errores.

HU14. Vista de errores: Debe mostrar un cuadro en el cual serán mostrados los errores para que se puedan corregir y así ejecutar el programa.

HU15. Ayuda: Se puede acceder a la ayuda desde la opción de la barra de menú y desde el icono de la barra de herramientas, en esta se encontrará lo necesario para utilizar la herramienta adecuadamente.

HU16. Estilos: Permite al editor cambiar el aspecto de la interfaz a diferentes estilos.

HU17. Traducción del Pseudocódigo: Se debe tener la opción de guardar el pseudocódigo a C#, C++ y Java.

HU18. Ejemplos: Se puede acceder a los ejemplos desde la opción del menú Archivo, en esta se encontrará los ejemplos aplicativos para utilizar en la herramienta del compilador.

ANEXO E

Instrumento de Evaluación del compilador de pseudocódigo.

ENCUESTA

INDICADORES DE USABILIDAD Y ERGONOMÍA DEL COMPILADOR DE PSEUDOCÓDIGO

NOTA: Seleccione la opción que en su opinión corresponda:

1. **¿Cómo usted califica la facilidad de uso sobre el funcionamiento del compilador de pseudocódigo?**
   1) Bueno ( )
   2) Regular ( )
   3) Malo ( )
2. **¿Con que nivel le ayudó realizar sus algoritmos mediante el uso del compilador de pseudocódigo?**
   1) Bueno ( )
   2) Regular ( )
   3) Malo ( )
3. **¿Cómo usted califica su adaptación en el desarrollo de algoritmos con el uso del compilador de pseudocódigo?**
   1) Bueno ( )
   2) Regular ( )
   3) Malo ( )
4. **¿Cómo le parece el diseño de la Interfaz del compilador de pseudocódigo?**
   1) Bueno ( )
   2) Regular ( )
   3) Mala ( )
5. **¿Cómo calificaría sobre el manejo intuitivo del compilador de pseudocódigo?**
   1) Bueno ( )
   2) Regular ( )
   3) Malo ( )
6. **¿Cree que el diseño del compilador de pseudocódigo es atractivo?**
   1) Bueno ( )
   2) Regular ( )
   3) Malo ( )

GRACIAS POR SU COLABORACIÓN

Instrumento de Recolección de datos: Registro de resolución de la encuesta según las preguntas que intervienen en los indicadores de usabilidad y ergonomía

	Indicadores de Usabilidad		Indicadores de Ergonomía			
Nro.	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**
1	Bueno	Bueno	Bueno	Bueno	Bueno	Bueno
2	Regular	Regular	Bueno	Bueno	Regular	Bueno
3	Regular	Bueno	Bueno	Bueno	Bueno	Bueno
4	Regular	Bueno	Regular	Bueno	Bueno	Bueno
5	Regular	Bueno	Regular	Regular	Regular	Bueno
6	Regular	Regular	Bueno	Bueno	Bueno	Regular
7	Regular	Regular	Bueno	Regular	Bueno	Bueno
8	Bueno	Regular	Bueno	Bueno	Bueno	Bueno
9	Bueno	Bueno	Regular	Regular	Bueno	Bueno
10	Bueno	Regular	Regular	Bueno	Bueno	Bueno
11	Bueno	Regular	Bueno	Regular	Bueno	Bueno
12	Bueno	Bueno	Bueno	Bueno	Bueno	Bueno
13	Bueno	Bueno	Regular	Bueno	Regular	Bueno
14	Bueno	Bueno	Bueno	Bueno	Bueno	Bueno
15	Bueno	Bueno	Bueno	Regular	Bueno	Regular
16	Regular	Regular	Regular	Bueno	Regular	Regular
17	Regular	Regular	Regular	Bueno	Regular	Regular
18	Bueno	Bueno	Bueno	Bueno	Bueno	Regular
19	Bueno	Bueno	Bueno	Regular	Bueno	Regular

Fuente: Elaboración propia

ANEXO F

Archivos de Pseudocódigo de Prueba

5. Suma de dos números

```
Inicio
    entero numero1, numero2,resultado;
    Escribir("Suma de dos numeros\n");
    Escribir("Ingrese numero 1:");
    Leer(numero1);
    Escribir("Ingrese numero 2:");
    Leer(numero2);
    resultado=numero1+numero2;
    Escribir("La suma de los numeros es:"+resultado);
Fin
```

6. Promedio de números

```
Inicio
    real cantidad, suma;
    real promedio;
    suma=0;
    Escribir("Promedio de los numeros\n");
    Escribir("Ingrese la cantidad de numeros:");
    Leer(cantidad);
    Para(entero i=0;i<cantidad;i++) Hacer
        Escribir("Ingrese numero "+(i+1)+" :");
        entero numeros;
        Leer(numeros);
        suma=suma+numeros;
    FinPara
    promedio=suma/cantidad;
    Escribir("Promedio de los numeros es:"+promedio);
Fin
```

7. Adivina número

```
Inicio
    entero numero, adivina;
    Escribir("++++++++++ADIVINA NUMERO++++++++++\n");
    adivina=Aleatorio(0,10);
    Repetir
        Escribir("Ingrese un numero entre 0 y 10:");
        Leer(numero);
        Si(numero==adivina) Entonces
            Escribir("Felicidades - Adivinaste el numero\n");
        Sino
            Escribir("Error - Fallaste\n");
        FinSi
    Hasta Que(adivina!=numero);
Fin
```

8. Factorial de un número

```
Inicio
        entero numero,resultado;
        Escribir("++++++++FACTORIAL DE UN NUMERO++++++++\n");
        Escribir("Ingrese un numero:");
        Leer(numero);
        resultado=Factorial(numero);
        Escribir("El factorial del numero "+ numero +" es: "+resultado);
Fin
Funcion entero Factorial(entero num)
        Si(num==0) Entonces
                retornar 1;
        Sino
                retornar num*Factorial(num-1);
        FinSi
FinFuncion
```

9. Número primo

```
Inicio
        entero n,i,contador;
        Escribir("Ingrese un numero:");
        Leer(n);
        contador=0;
        Para(i=1;i<=n;i++) Hacer
                        Si(n%i==0) Entonces
                                contador=contador+1;
                        FinSi
        FinPara
        Si(contador==2) Entonces
                Escribir("El numero es primo");
        Sino
                Escribir("El numero no es primo");
        FinSi
Fin
```

ANEXO G

DESCRIPCIÓN DEL LENGUAJE HITO

Lenguaje HITO

Se denomina Lenguaje HITO al lenguaje utilizado para la introducción de los estudiantes al mundo de la programación en la asignatura de Algorítmica I, la cual es impartida como parte del programa de la carrera profesional de Ingeniería Informática y Sistemas de la Universidad Nacional Micaela Bastidas de Apurímac.

A continuación se presentan los diagramas de sintaxis que describen la estructura de un programa escrito en este lenguaje así como las características particulares del mismo.

Diagramas de sintaxis

Estructura general de un algoritmo escrito en el lenguaje HITO:

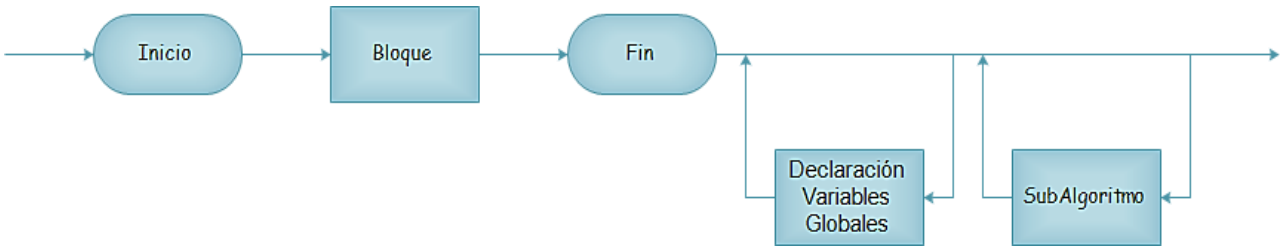


Figura N° 35 : Diagrama de sintaxis de la estructura general de un algoritmo

Fuente: Elaboración propia

Declaración Variable Global:

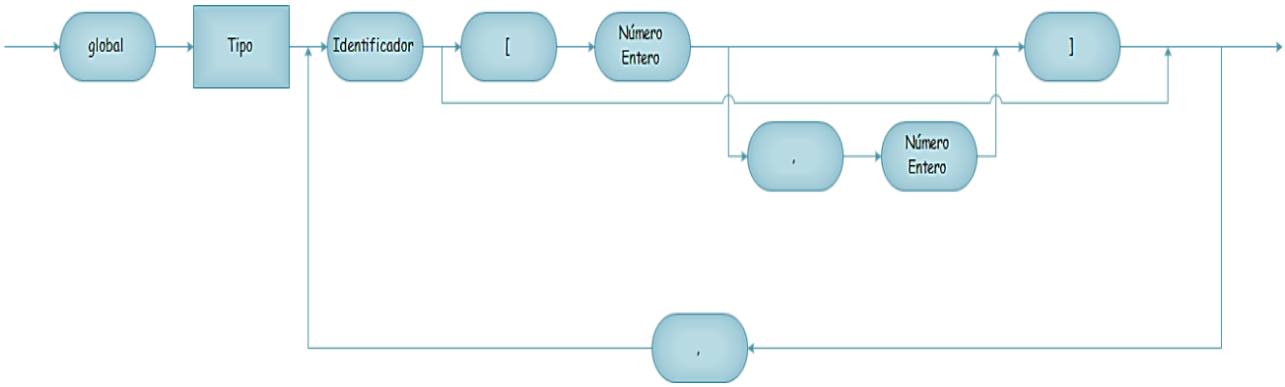


Figura N° 36: Diagrama de sintaxis para la declaración de una variable global

Fuente: Elaboración propia

Declaración Variable:

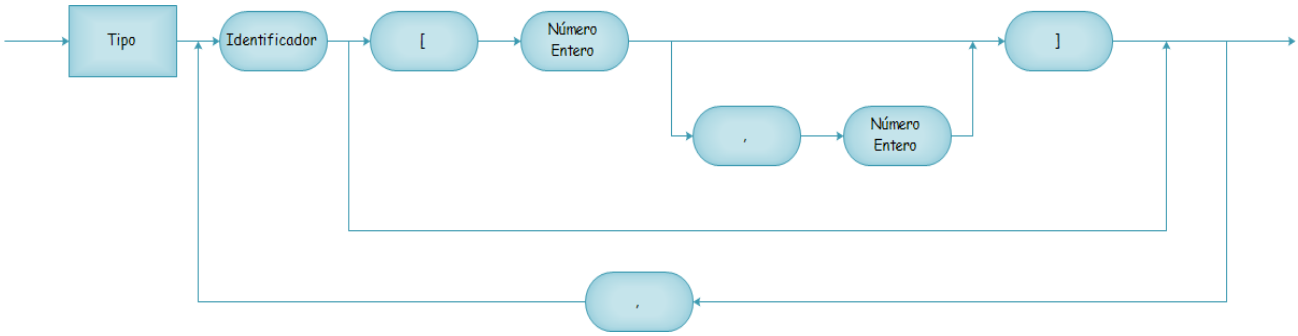


Figura N° 37: Diagrama de sintaxis para la declaración de una variable

Fuente: Elaboración propia

Tipo:

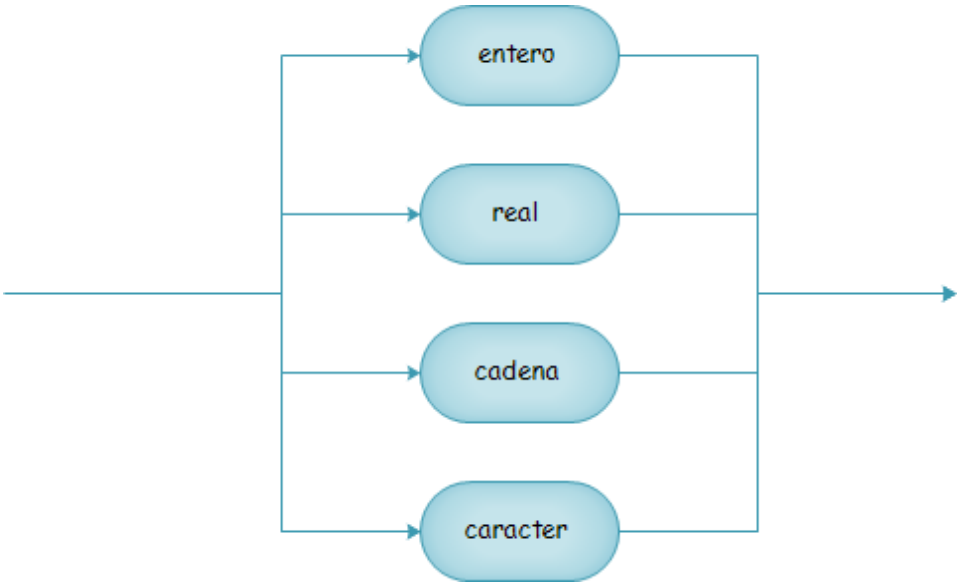


Figura N° 38 : Diagrama de sintaxis que muestra los tipos existentes en el lenguaje HITO

Fuente: Elaboración propia

Subalgoritmo:

Figura N° 39 : Diagrama de sintaxis para estructura de un subalgoritmo

Fuente: Elaboración propia

Declarar Sub Algoritmo:

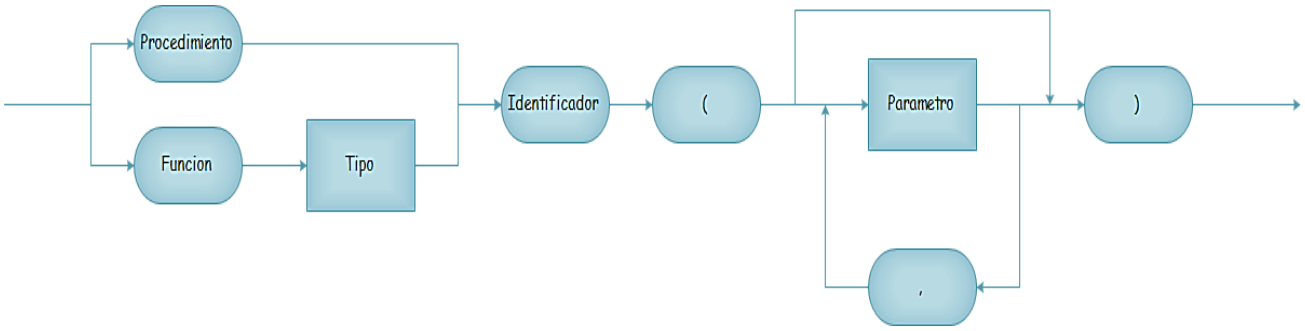


Figura N° 40 : Diagrama de sintaxis para la firma de los subalgoritmos

Fuente: Elaboración propia

Parámetro:

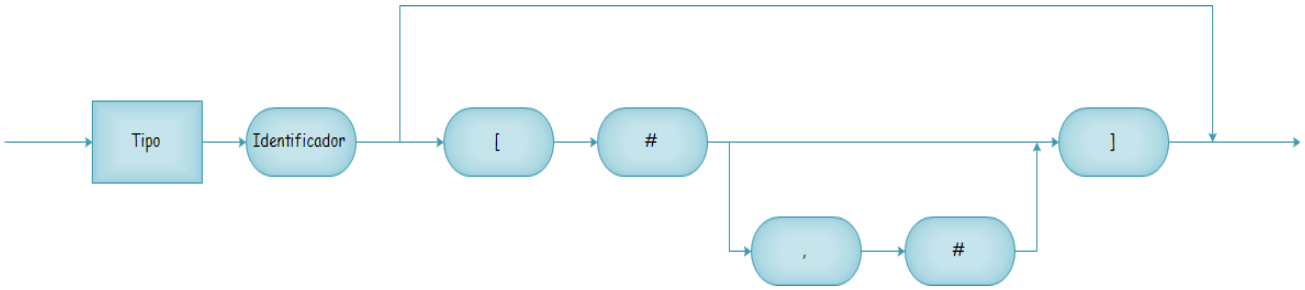


Figura N° 41 : Diagrama de sintaxis para los parámetros de subalgoritmos

Fuente: Elaboración propia

Variable:

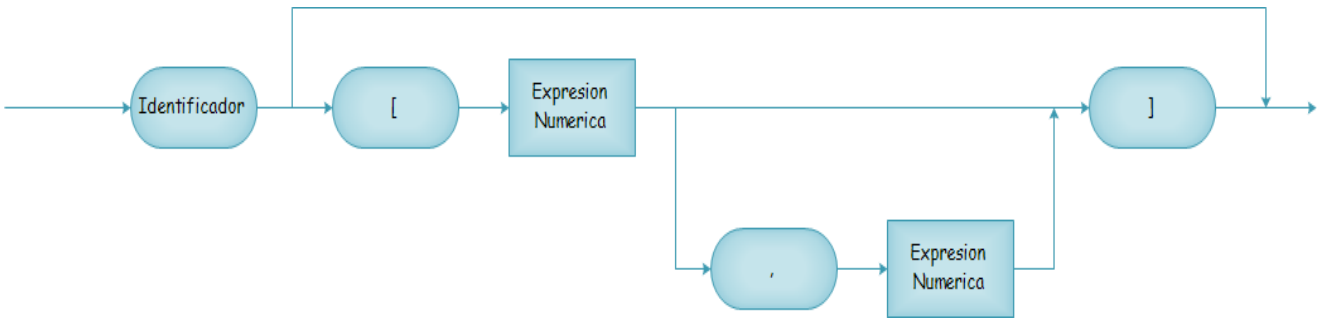


Figura N° 42 : Diagrama de sintaxis que muestra la forma en que puede utilizarse una variable dependiendo de si es un arreglo, o matriz o variable sencilla

Fuente: Elaboración propia

Bloque:

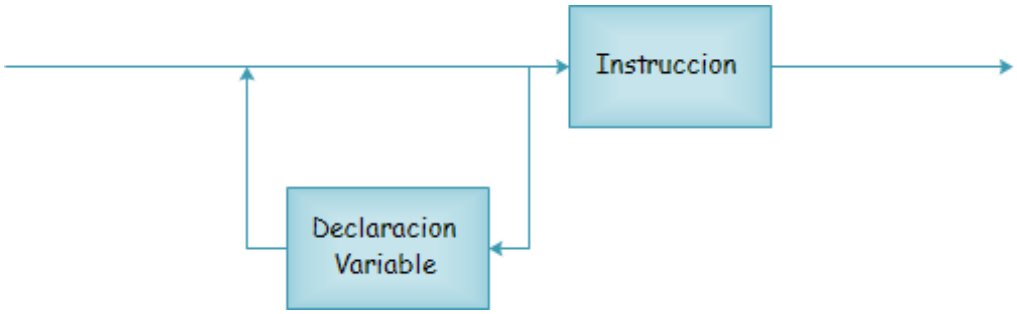


Figura N° 43 : Diagrama de sintaxis para la estructura interna de un bloque en el subalgoritmo principal y en los subalgoritmos externos

Fuente: Elaboración propia

Instrucciones E/S:

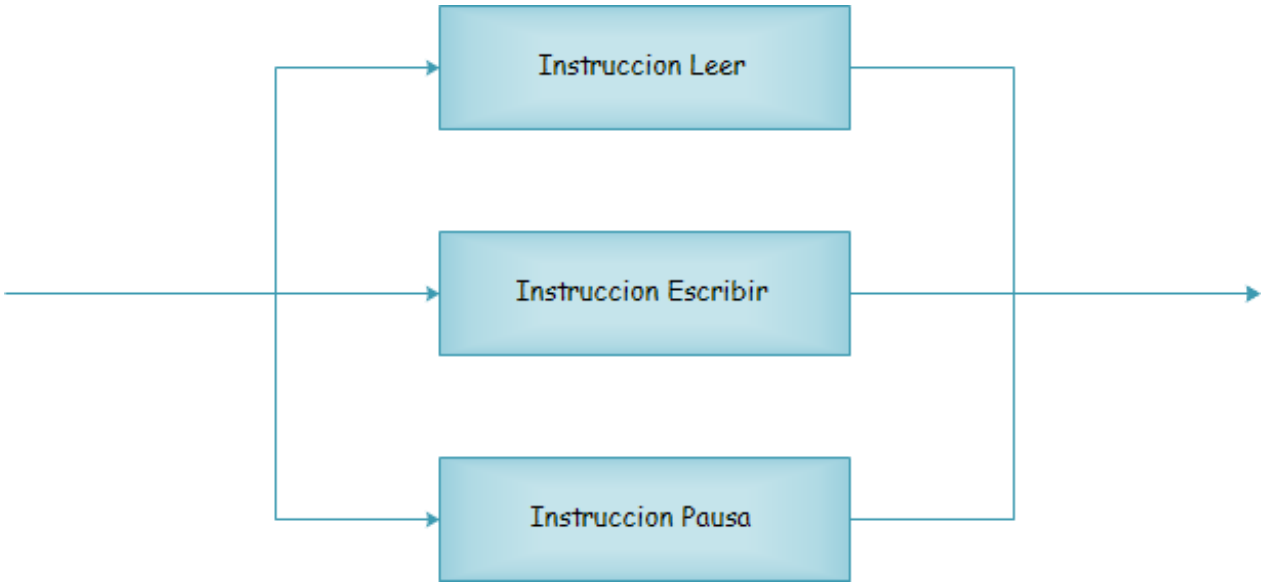


Figura N° 44 : Diagrama de sintaxis instrucciones de entrada y salida de datos

Fuente: Elaboración propia

Instrucción:

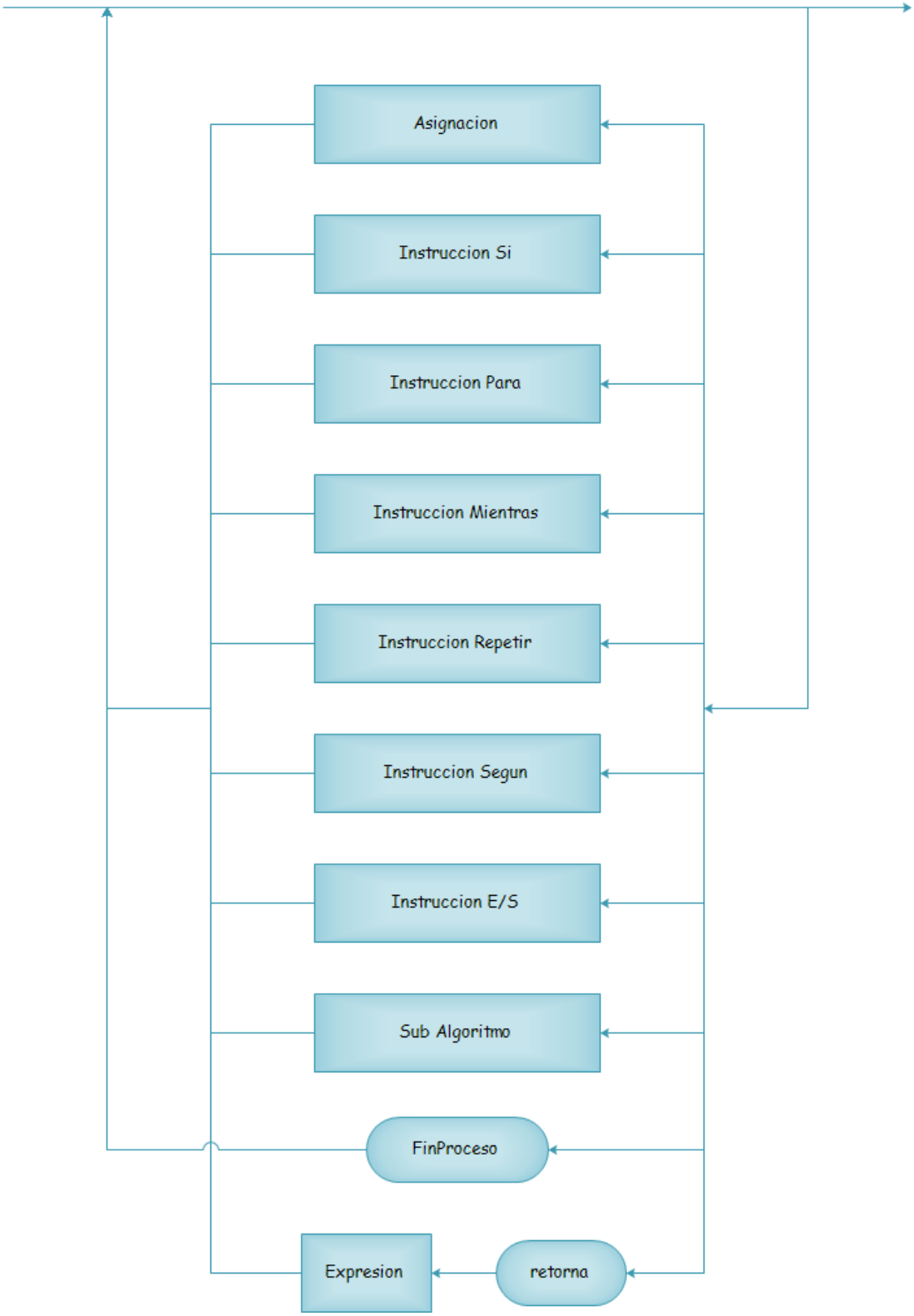


Figura N° 45 : Diagrama de sintaxis que muestra las diferentes instrucciones que pueden utilizarse en un algoritmo

Fuente: Elaboración propia

Llamada_Subalgoritmo:

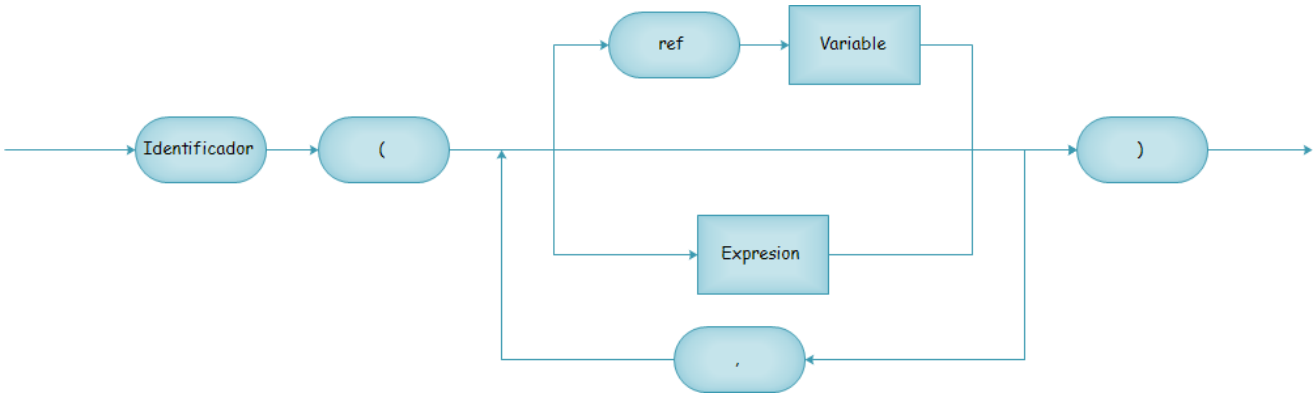


Figura N° 46 : Diagrama de sintaxis para la instrucción de llamada a un subalgoritmo

Fuente: Elaboración propia

Asignacion:

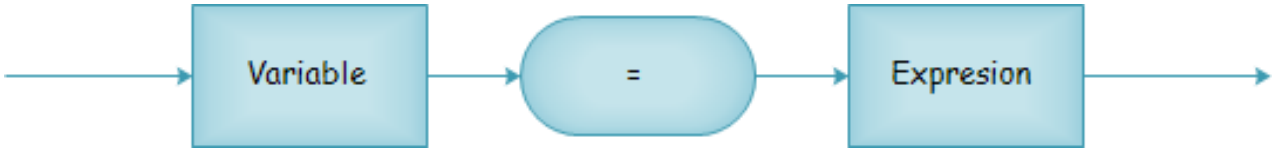


Figura N° 47 : Diagrama de sintaxis de instrucción de asignación de un valor a una variable

Fuente: Elaboración propia

Instrucción Si:

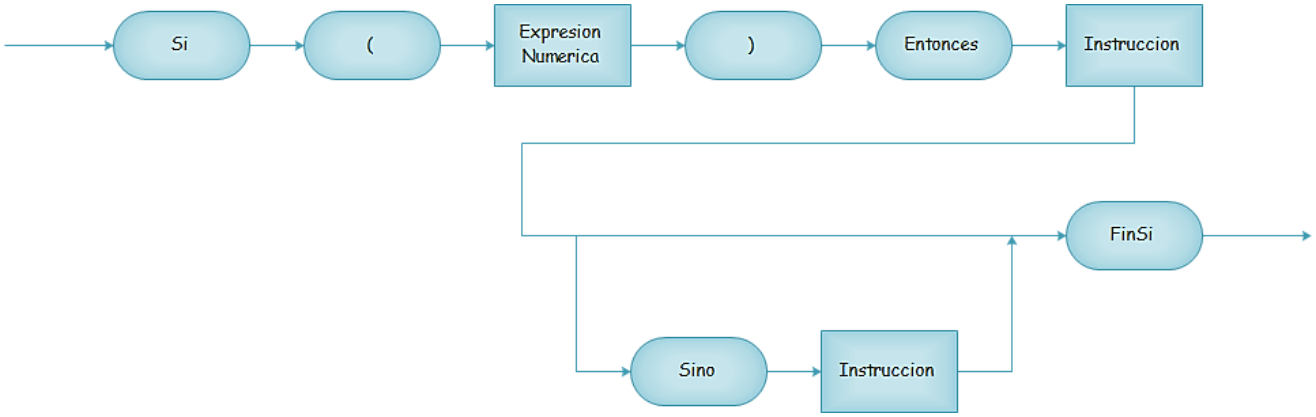


Figura N° 48 : Diagrama de sintaxis para la instrucción Si

Fuente: Elaboración propia

Instruccion_Segun:

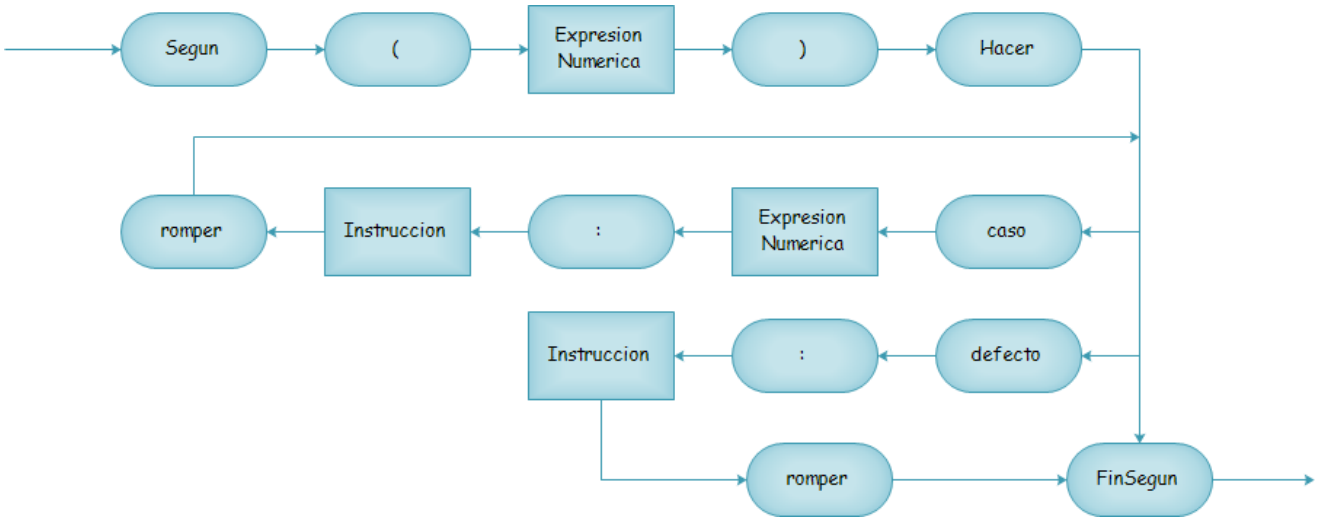


Figura N° 49 : Diagrama de sintaxis para la instrucción Segun

Fuente: Elaboración propia

Instrucción Mientras:

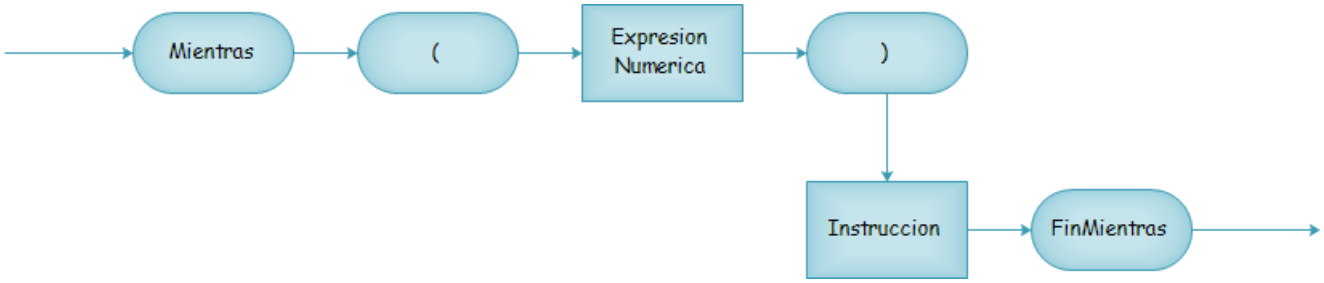


Figura N° 50 : Diagrama de sintaxis para la instrucción mientras

Fuente: Elaboración propia

Instruccion_Repetir:

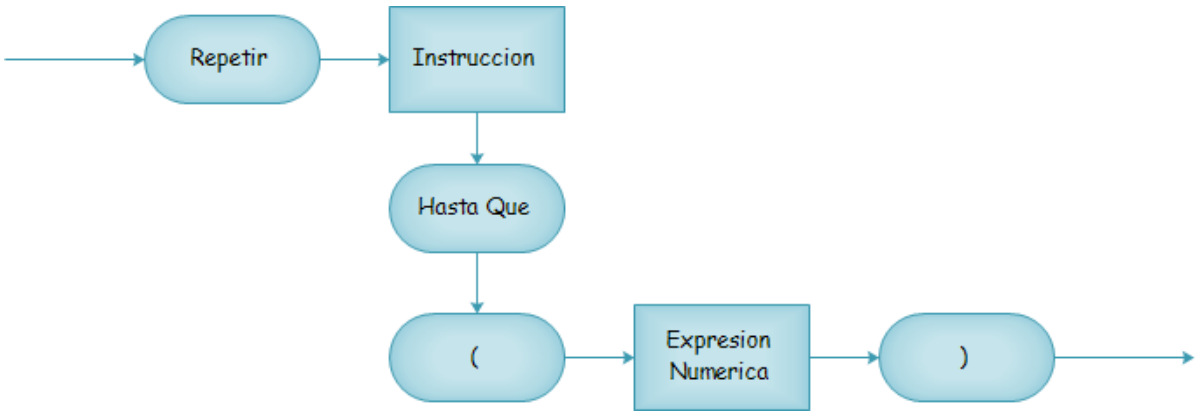


Figura N° 51 : Diagrama de sintaxis para la instrucción repetir

Fuente: Elaboración propia

Instruccion_Para:

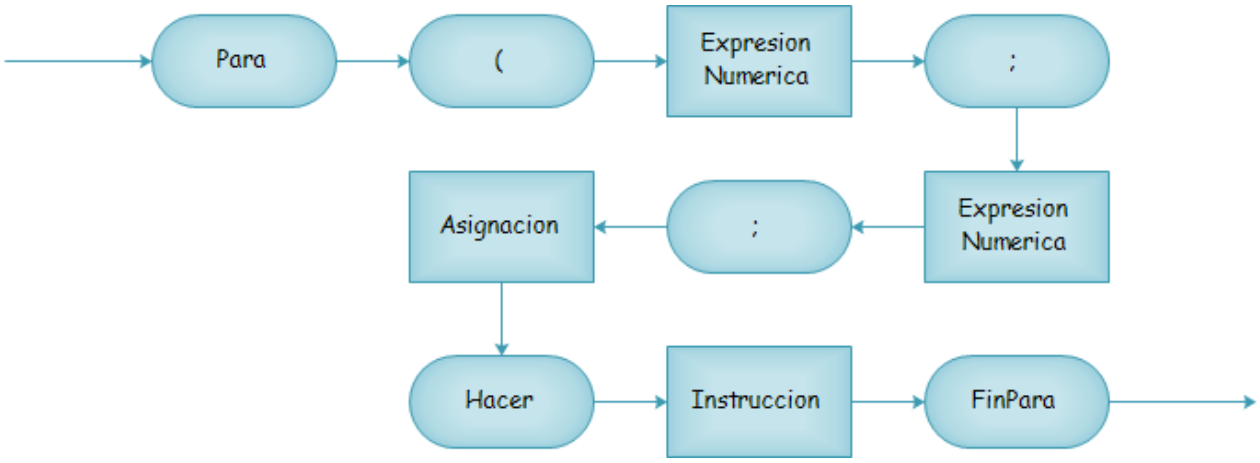


Figura N° 52 : Diagrama de sintaxis para la instrucción para

Fuente: Elaboración propia

Instruccion_Leer:

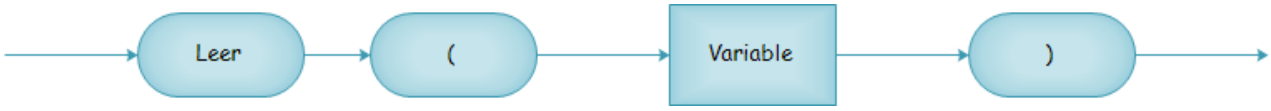


Figura N° 53 : Diagrama de sintaxis para la instrucción leer

Fuente: Elaboración propia

Instruccion_Escribir:

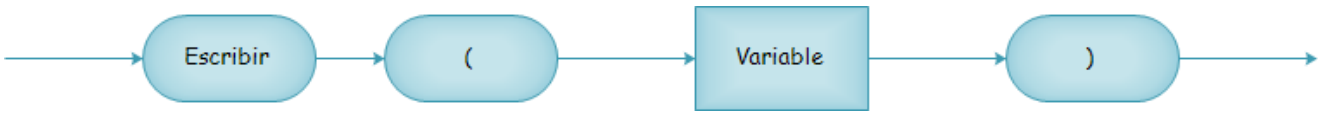


Figura N° 54 : Diagrama de sintaxis para la instrucción escribir

Fuente: Elaboración propia

Instruccion_Pausa:

Figura N° 55 : Diagrama de sintaxis para la instrucción Pausa

Fuente: Elaboración propia

Expresion:

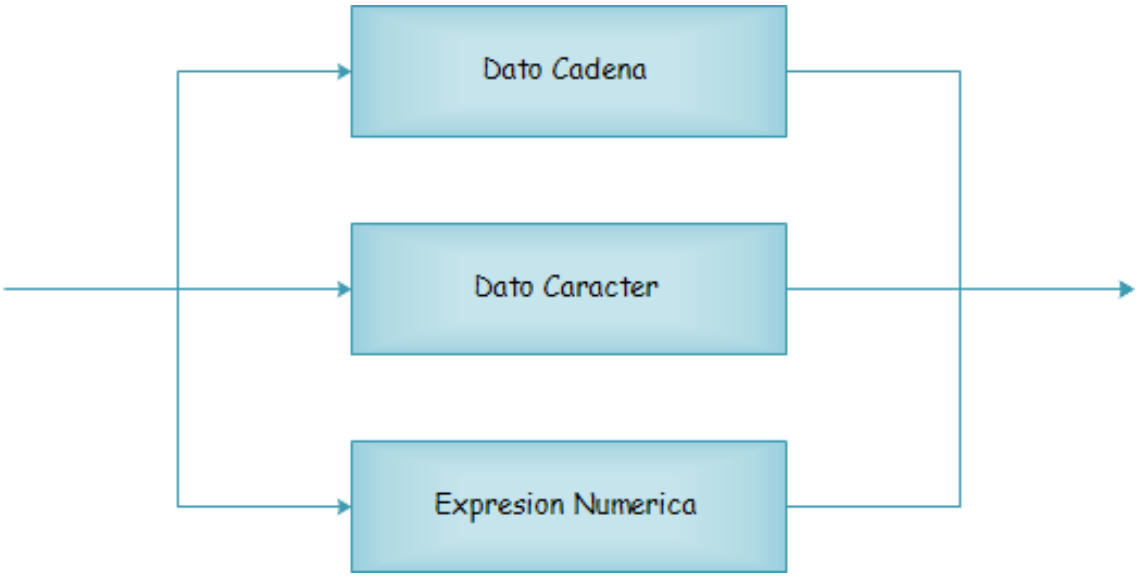


Figura N° 56 : Diagrama de sintaxis de las diferentes expresiones

Fuente: Elaboración propia

Expresion_Numerica:

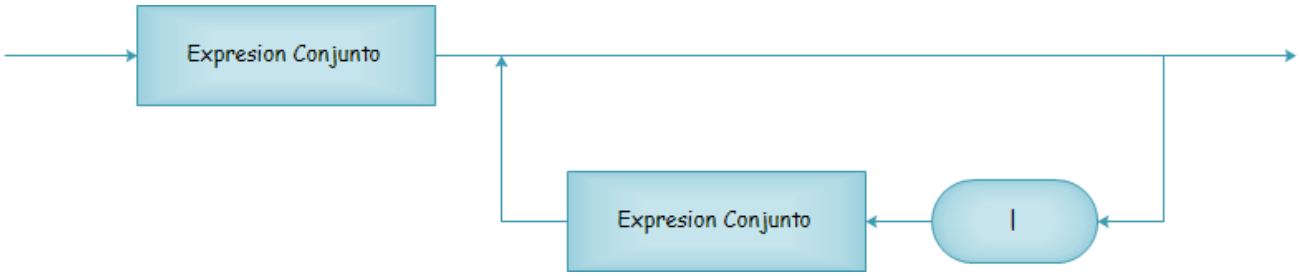


Figura N° 57 : Diagrama de sintaxis para una expresión numérica

Fuente: Elaboración propia

Expresion_Conjuncion:

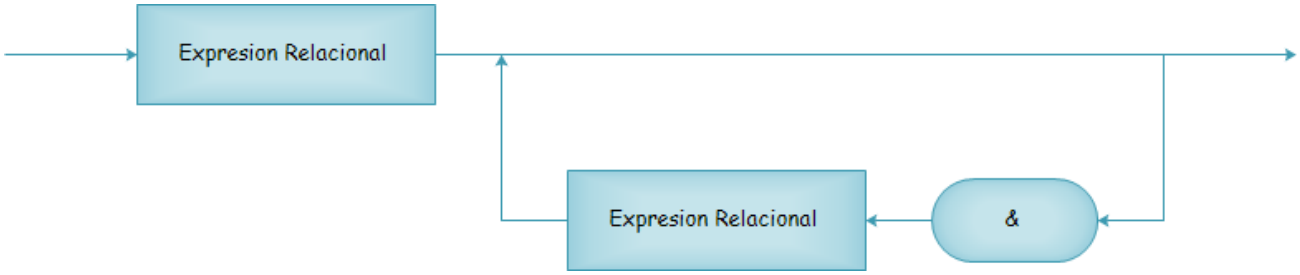


Figura N° 58 : Diagrama de sintaxis para una expresión de conjunción

Fuente: Elaboración propia

Expresion_Relacional:

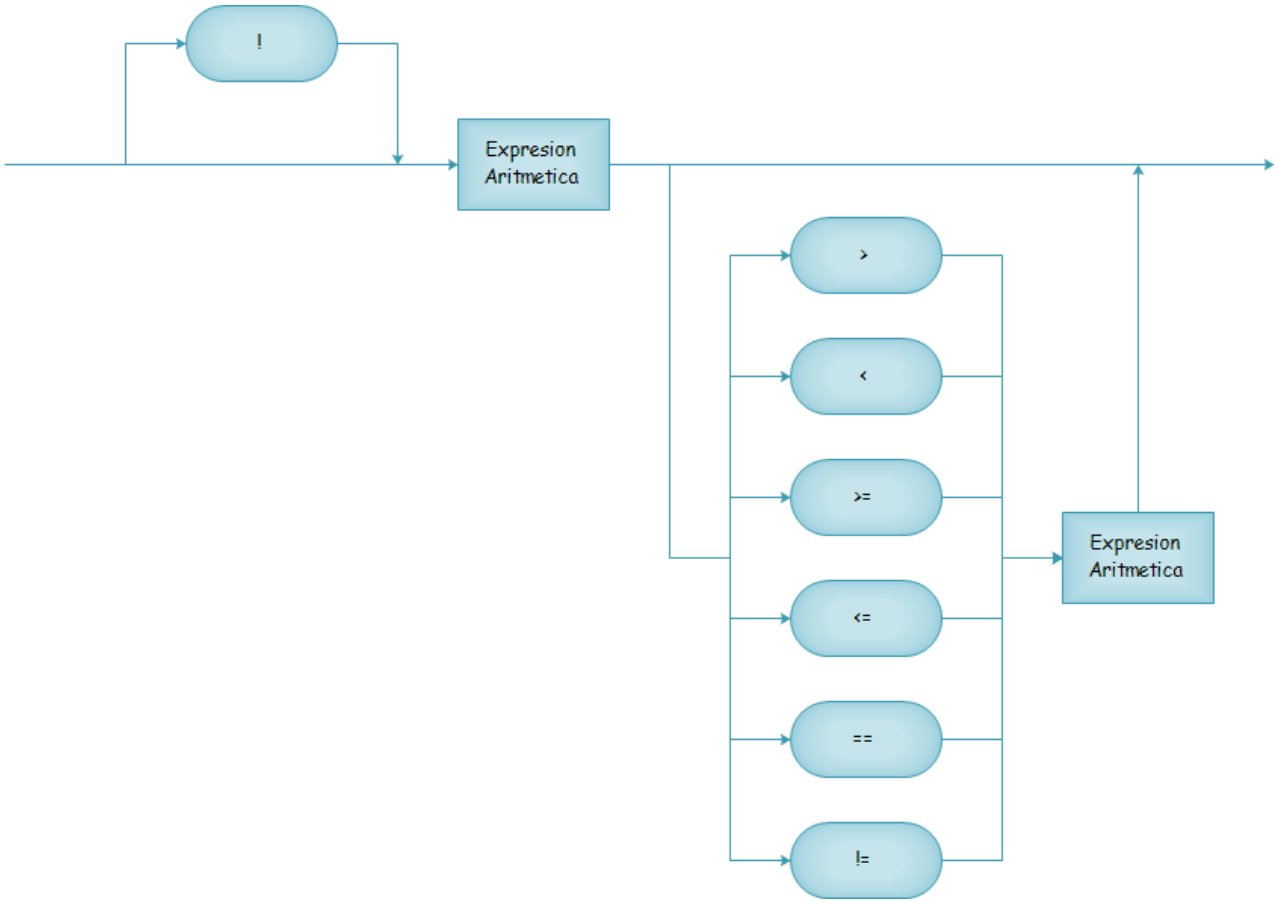


Figura N° 59 : Diagrama de sintaxis para una expresión relacional

Fuente: Elaboración propia

Expresion_Aritmetica:

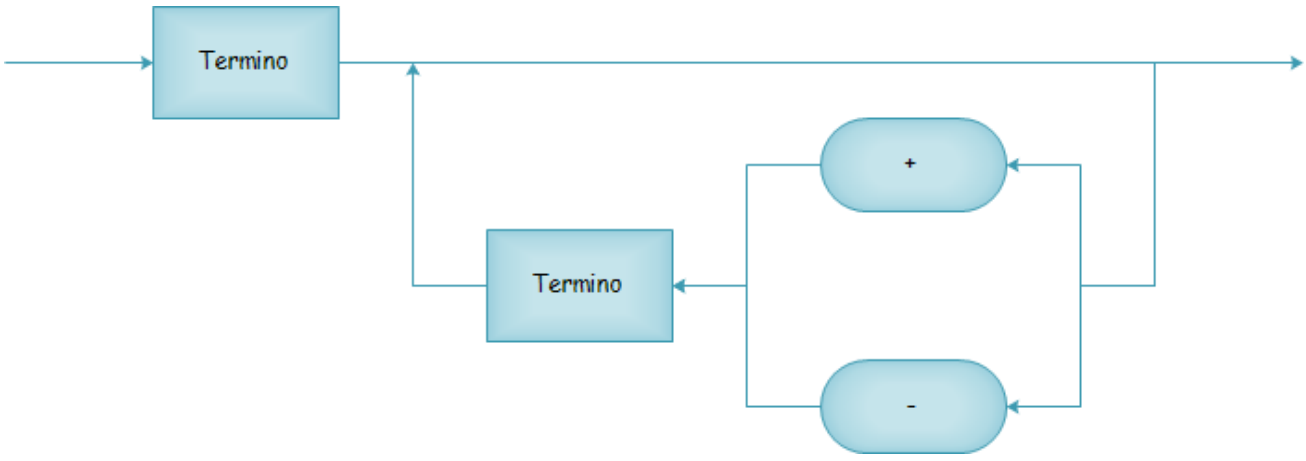


Figura N° 60 : Diagrama de sintaxis para la estructura de una expresión aritmética

Fuente: Elaboración propia

Termino:

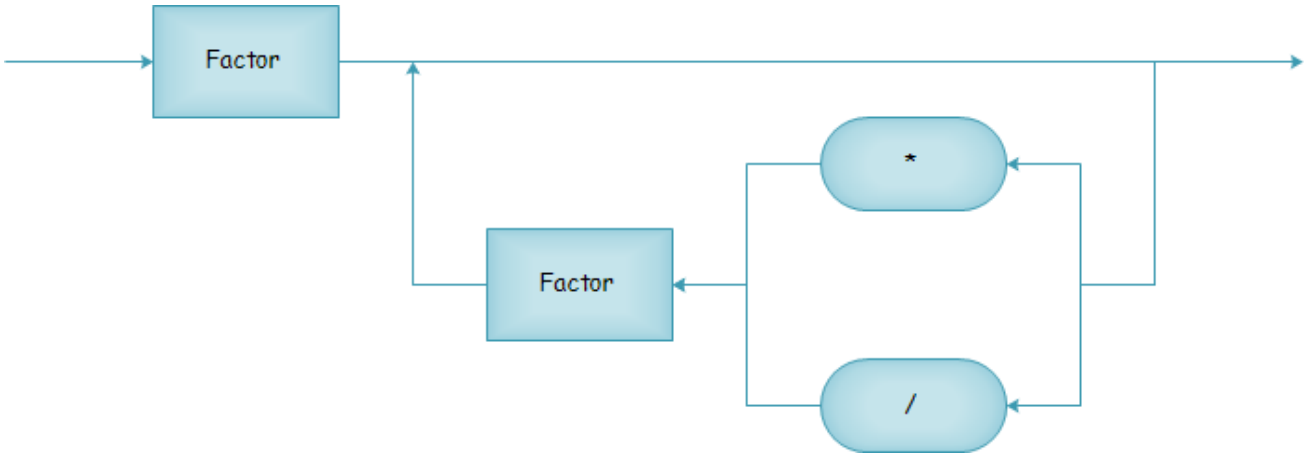


Figura N° 61 : Diagrama de sintaxis de un término en una expresión numérica

Fuente: Elaboración propia

Factor:

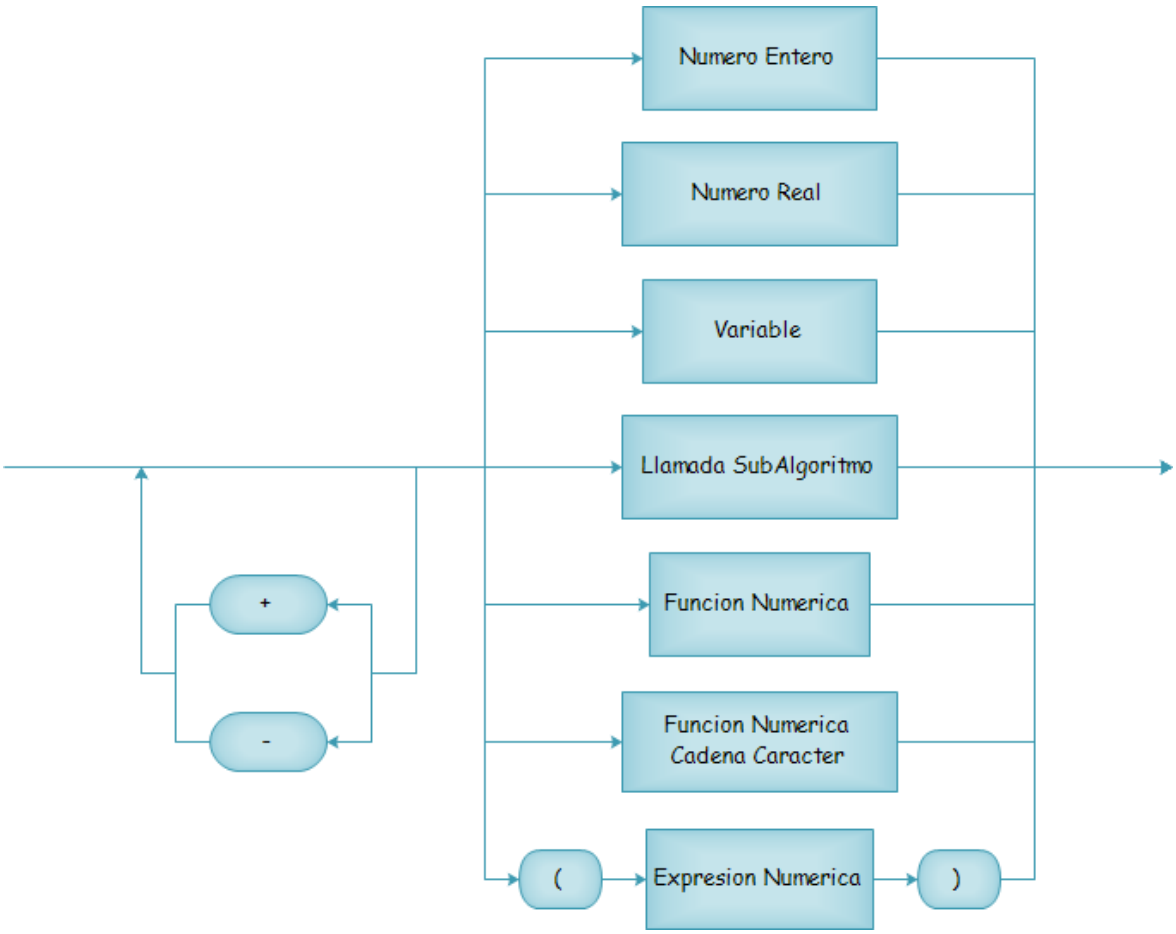


Figura N° 62 : Diagrama de sintaxis que muestra los elementos que pueden conformar un factor en una expresión numérica

Fuente: Elaboración propia

Función Cadena:

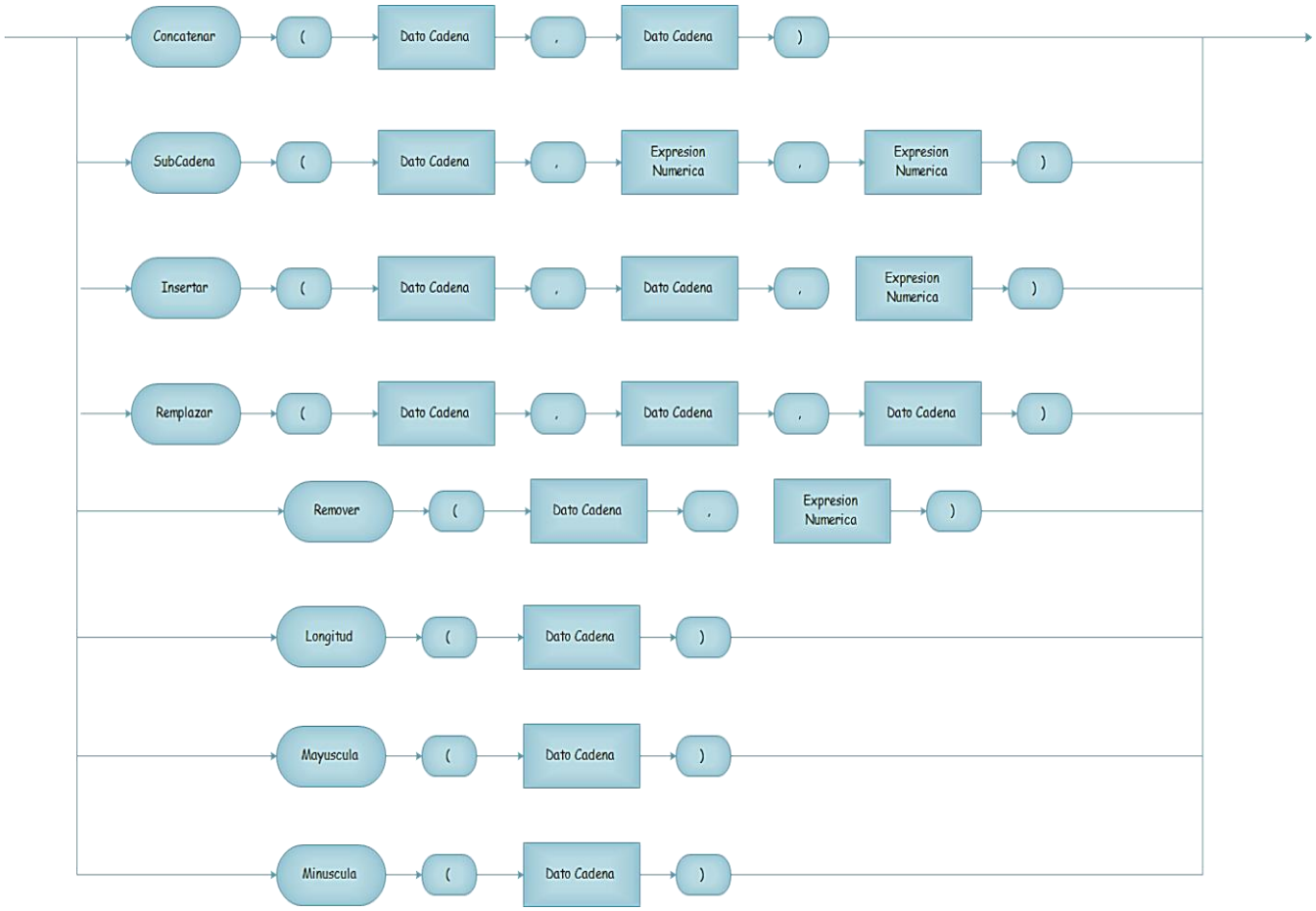


Figura N° 63 : Diagrama de sintaxis que muestra la estructura de las funciones que devuelven una cadena trabajando con cadenas

Fuente: Elaboración propia

Dato Caracter:

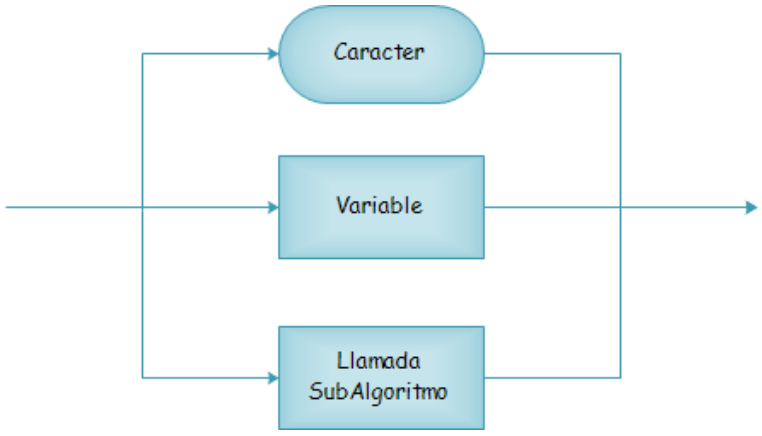


Figura N° 64 : Diagrama de sintaxis para los datos de tipo caracter utilizados como parámetro en subalgoritmos o como valores en expresiones

Fuente: Elaboración propia

Función Numérica:

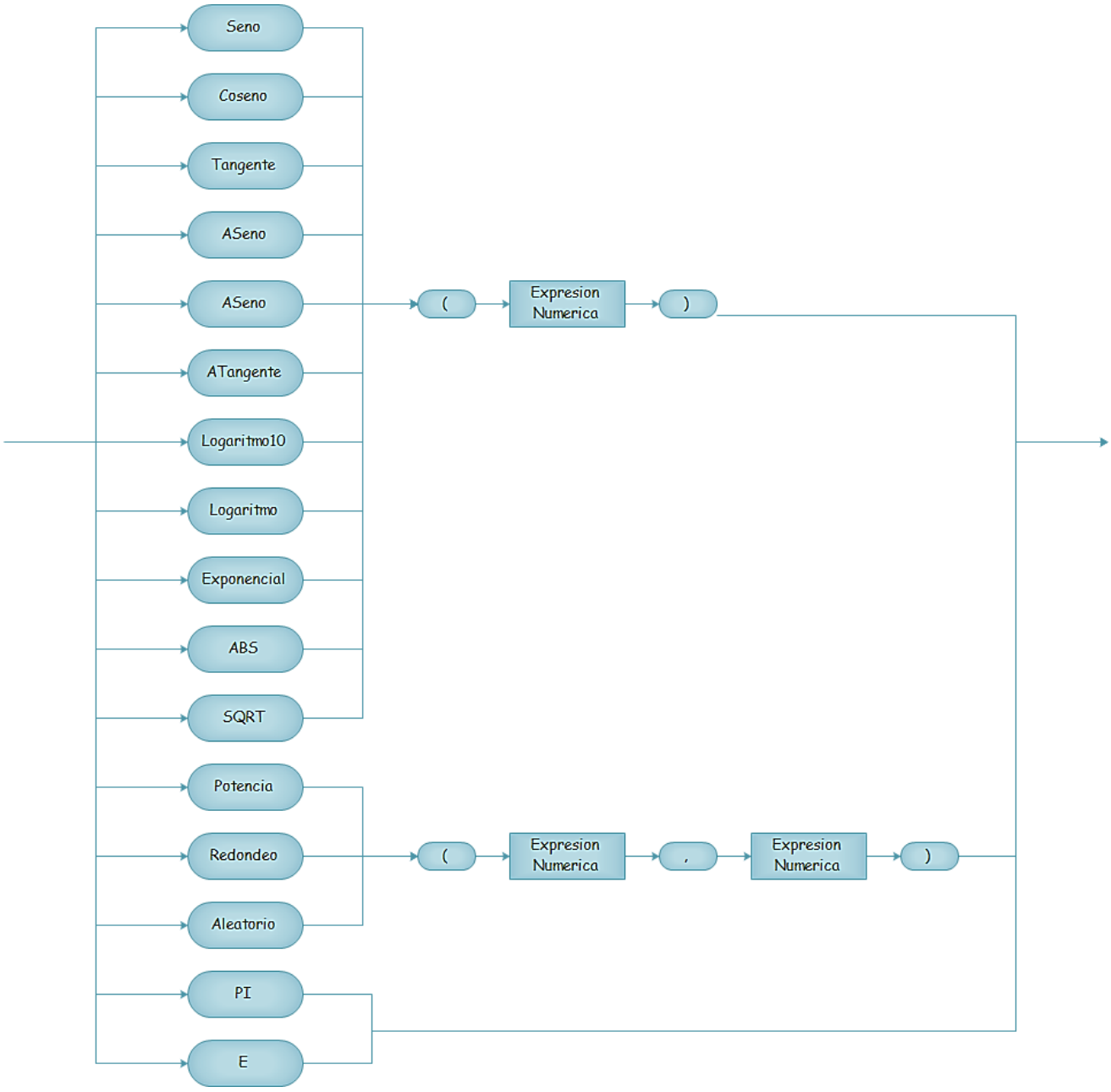


Figura N° 65 : Diagrama de sintaxis que muestra la estructura de las funciones que devuelven un valor numérico trabajando con expresiones numéricas

Fuente: Elaboración propia

Dato Cadena:

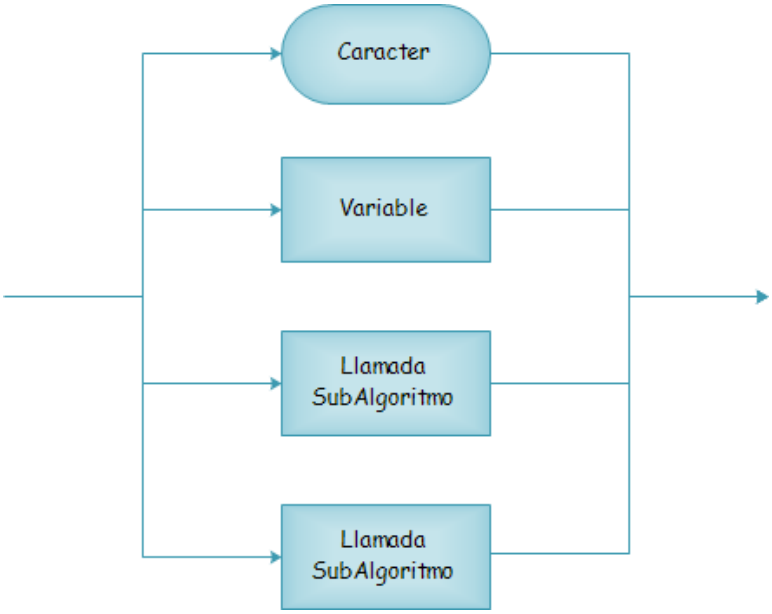


Figura N° 66 : Diagrama de sintaxis para los datos de tipo cadena utilizados como parámetro en subalgoritmos o como valores en expresiones

Fuente: Elaboración propia

Tipos de datos

El lenguaje HITO soporta los siguientes tipos de datos:

Tabla N° 11 : Tipos de datos existentes en HITO

Tipo	Descripción
Entero	Número entero positivo o negativo de hasta 30 cifras.
Real	Número real positivo o negativo de hasta 30 cifras enteras con un máximo de 20 cifras decimales.
Carácter	Caracter ASCII simple.
Cadena	Secuencia de hasta 3000 caracteres ASCII simples.

Fuente: Elaboración propia

Operadores y funciones

El lenguaje de HITO cuenta con los siguientes tipos de operadores:

1. **Operadores Aritméticos**

Tabla N° 12: Operadores aritméticos

Operación	Representación en HITO
Suma	+
Resta	-
Multiplicación	*
División	/

Fuente: Elaboración propia

2. **Operadores Relacionales**

Tabla N° 13: Operadores relacionales

Operación	Representación en HITO
Menor que	<
Menor o igual que	<=
Mayor que	>
Mayor o igual que	>=
Comparación de igualdad	==
Diferente	¡=

Fuente: Elaboración propia

3. **Operadores Lógicos**

Tabla N° 14 : Operadores Lógicos

Operación	Representación en HITO
Y Lógico	&&
O Lógico	||
Negación	¡

Fuente: Elaboración propia

4. Otros operadores y caracteres especiales

Tabla N° 15 : Otros operadores y caracteres especiales

Operación	Representación en HITO
Asignación	=
Concatenación (para elementos en las instrucciones escribir)	+
Comentario de código (una sola línea)	//
Comentario de código (bloque de código)	/* */

Fuente: Elaboración propia

En cuanto a funciones predefinidas del lenguaje HITO se pueden mencionar dos tipos:

5. Funciones numéricas

Tabla N° 16 : Funciones numéricas

Operación	Sintaxis en HITO	Tipo de dato devuelto
Obtener el valor absoluto de un número	ABS(x)	real
Obtener el arco coseno	Acos(x)	real
Obtener el arco seno	Asin(x)	real
Obtener el arco tangente	Atan(x)	real
Evaluar el coseno de un número	Coseno(x)	real
Evaluar el exponencial de un numero	Exponencial(x):	real
Obtener el logaritmo natural o neperiano	Logaritmo(x)	real
Obtener el logaritmo base 10 de un número	Logaritmo10(x)	real
Evaluar un número a la potencia	Potencia(x,y)	real
Redondear un numero	Redondeo(x, cifras)	real
Obtener el seno de un número	Seno(x)	real
Evaluar la raíz cuadrada de un número	SQRT(x):	real
Evaluar la tangente de un número	Tan(x)	real
Obtener el valor del número π	PI	real
Obtener el valor del número E	E	real

Fuente: Elaboración propia

6. Funciones para el manejo de datos de tipo cadena o carácter

Tabla N° 17 : Funciones para el manejo de datos de tipo cadena o carácter

Operación	Sintaxis en HITO	Tipo de dato devuelto
Comparar dos cadenas	Comparar(cadena 1, cadena 2)	booleano
Obtener la longitud de una cadena	Longitud(cadena)	entero
Concatenar dos cadenas	Concatenar(cadena 1, cadena 2)	cadena
Extraer una sub-cadena de una cadena	SubCadena(cadena, liminf, limsup)	cadena
Insertar una cadena dentro de otra cadena	Insertar(Cadena, CadenaInsertar, Posición):	cadena
Elimina una cantidad de caracteres en cierta posición de una cadena	Remover(CadenaTexto, Posición):	cadena
Sustituye una cadena (todas las veces que aparezca) por otra cadena	Remplazar(CadenaTexto, CadenaBusqueda, CadenaRemplazar):	cadena
Convertir los caracteres de una cadena en mayúsculas	Mayuscula (cadena)	cadena
Convertir los caracteres de una cadena en minúsculas	Minuscula (cadena)	cadena

Fuente: Elaboración propia

7. Palabras reservadas

Tabla N° 18 : Palabras reservadas

entero	real	cadena	caracter
vacio	booleano	constante	global
nuevo	referencia	verdadero	falso
Inicio	Fin	FinProceso	Leer
Escribir	Escribir	Si	Entonces
Sino	FinSi	Segun	Hacer
caso	romper	continue	defecto
FinSegun	Mientras	FinMientras	Repetir

Hasta Que	Para	FinPara	Funcion
retornar	FinFuncion	Procedimiento	FinProcedimiento
ABS	Redondeo	SQRT	Seno
Coseno	Tangente	ASeno	ACoseno
ATangente	Logaritmo10	Logaritmo	Exponencial
Potencia	Aleatorio	ConvertirEntero	ConvertirReal
ConvertirCadena	ConvertirCaracter	Mayuscula	Minuscula
Longitud	SubCadena	Concatenar	Insertar
Remover	Reemplazar	Comparar	Tiempo
DibujarPunto	DibujarLinea	DibujarImagen	DibujarTexto
DibujarRectangulo	DibujarArco	PI	E

Fuente: Elaboración propia

ANEXO H

GUÍA DE USUARIO

MANUAL HITO

Juan Carlos Muñoz Miranda

Universidad Nacional Micaela Bastidas de Apurímac

HITO
Manual de Programación
Juan Carlos Muñoz Miranda

ÍNDICE

Bienvenido a Hito

Hito es un software para desarrollo de aplicaciones en pseudocódigo, una herramienta para iniciar en el mundo de la programación. Su objetivo es de permitirle el aprendizaje de pseudocódigo en español, proveyéndole un rico conjunto de bibliotecas para tareas comunes, así como y una interfaz intuitiva y sencilla.

¿Para quién es Hito?

Además del análisis y la construcción de algoritmos, influye en gran nivel en las personas que deseen iniciar en el mundo de la programación; fundamental para el aprendizaje de la lógica informática. Del mismo modo encontramos la herramienta de desarrollo de pseudocódigo en español llamado PSeInt; la cual no provee de las funcionalidades necesarias para el aprendizaje en el desarrollo modular de pseudocódigos; es decir la capacidad de poder crear tanto funciones como procedimientos que permita dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

1. Toma de contacto con HITO

Hito es un lenguaje de pseudocódigo en español cuya sintaxis está basado en el lenguaje de programación C#. Se trata de un lenguaje de pseudocódigo moderno, el cual provee de diferentes funcionalidades para el desarrollo de aplicaciones en pseudocódigo.

Se trata de un lenguaje que permite crear programas para la plataforma .NET, y del que existe una funcionalidad de poder exportar el pseudocódigo a diferentes lenguajes de programación como C#, Java y C++.

Comenzaremos por usar la plataforma de desarrollo durante los primeros temas. Cuando los conceptos básicos estén asentados, pasaremos a desarrollar aplicaciones más complejas.

Los pasos que seguiremos para crear un programa en pseudocódigo serán:

1. Escribir el pseudocódigo (fichero fuente) en nuestro editor.
2. Compilarlo con nuestro compilador. Esto creará un "fichero ejecutable".
3. Lanzar el fichero ejecutable.

El compilador permite dar todos estos pasos desde un único entorno, en el que escribimos nuestros programas, los compilamos, y los depuramos en caso de que exista algún fallo.

1.1. Escribir el hola mundo

Vamos con un primer ejemplo, posiblemente el más sencillo de todos, escribir el tradicional “Hola Mundo” en pantalla.

```
Inicio
        Escribir("Hola Mundo");
Fin
```

Esto escribe "Hola Mundo" en la pantalla. Pero hay mucho alrededor de ese "Hola Mundo", y vamos a comentarlo antes de proseguir, aunque muchos de los detalles se irán aclarando más adelante.

En este primer análisis, iremos de dentro hacia fuera:

- Escribir("Hola Mundo"); "Hola Mundo" es el texto que queremos escribir, y Escribir es la orden encargada de escribir una línea de texto en pantalla.
- Escribir("Hola Mundo"); porque Escribir es una orden de manejo de la "consola" (la pantalla "negra" en modo texto del sistema operativo).
- Inicio; indica cual es "el cuerpo del programa", la parte principal (un programa puede estar dividido en varios fragmentos, como veremos más adelante). Todos los programas tienen que tener un bloque principal de Inicio y Fin.

1.2. Cómo probar este programa

Como ya hemos comentado, usaremos Hito como plataforma de desarrollo para nuestros primeros programas. Por eso, vamos a comenzar por ver dónde encontrar esta herramienta, cómo instalarla y cómo utilizarla.

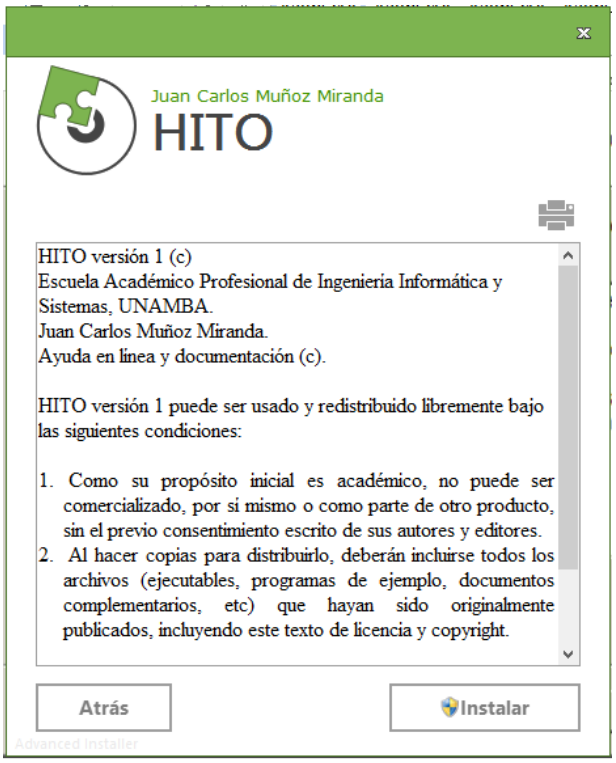
Podemos descargar Hito desde su página oficial:

https://www.facebook.com/hitojcmm/

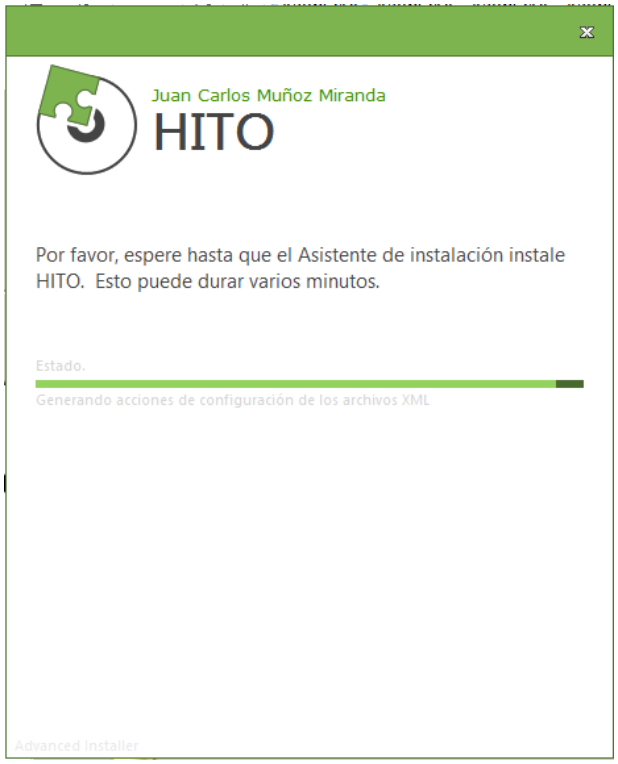
En la parte superior derecha aparece el enlace para descargar, que nos lleva a una nueva página para seleccionar los servidores web disponibles para descarga directa.

Se trata de un fichero de cerca de 16 Mb. Cuando termina la descarga, haremos doble clic en el fichero recibido, aceptaremos el aviso de seguridad que posiblemente nos mostrará Windows, y comenzará la instalación, en la que primero se nos muestra el mensaje de bienvenida:

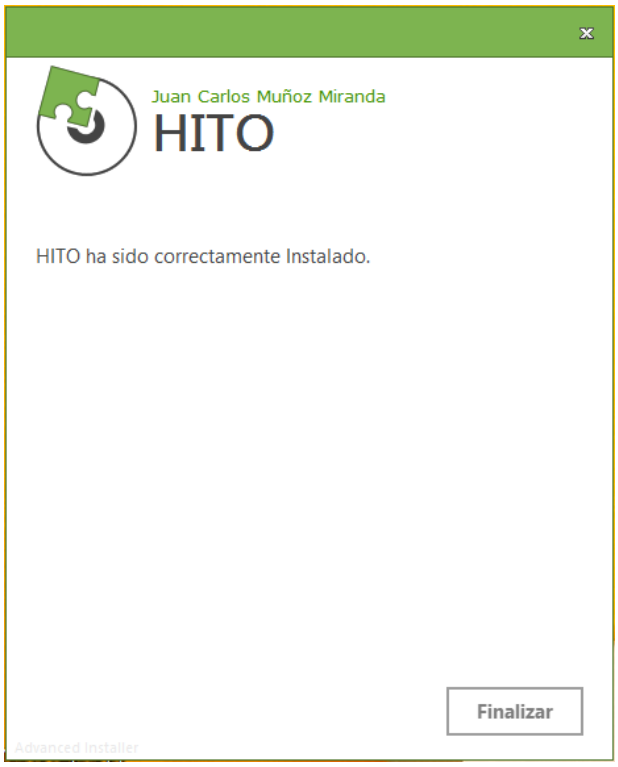
El siguiente paso será aceptar el acuerdo de licencia:



Luego presionamos el botón instalar y comienza la copia de ficheros:



Si todo es correcto, al cabo de un instante tendremos el mensaje de confirmación de que la instalación se ha completado:



1.3. Mostrar números enteros en pantalla

Cuando queremos escribir un texto "tal cual", como en el ejemplo anterior, lo encerramos entre comillas. Pero no siempre querremos escribir textos prefijados. En muchos casos, se tratará de algo que habrá que calcular.

El ejemplo más sencillo es el de una operación matemática. La forma de realizarla es sencilla: no usar comillas en el comando Escribir. Entonces, el compilador intentará analizar el contenido para ver qué quiere decir. Por ejemplo, para sumar 2 y 3 bastaría hacer:

```
Inicio
        Escribir(2+3);
Fin
```

1.4. Operaciones aritméticas básicas

1.4.1. Operadores

Está claro que el símbolo de la suma será un +, y podemos esperar cual será el de la resta, pero alguna de las operaciones matemáticas habituales tienen símbolos menos intuitivos. Veamos cuales son los más importantes:

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de la división(Modulo)

1.4.2. Orden de prioridad de los operadores

- En primer lugar se realizarán las operaciones indicadas entre paréntesis.
- Luego la negación.
- Después las multiplicaciones, divisiones y el resto de la división.
- Finalmente, las sumas y las restas.
- En caso de tener igual prioridad, se analizan de izquierda a derecha.

1.5. Introducción a las variables

Las variables son algo que no contiene un valor predeterminado, un espacio de memoria al que nosotros asignamos un nombre y en el que podremos almacenar datos. El primer ejemplo nos permitía escribir "Hola Mundo". El segundo nos permitía sumar dos números que habíamos prefijado en nuestro programa.

Por eso necesitaremos usar variables, zonas de memoria en las que guardemos los datos con los que vamos a trabajar y también los resultados temporales.

1.5.1. Definición de variables: números enteros

Para usar una cierta variable primero hay que declararla: indicar su nombre y el tipo de datos que querremos guardar.

El primer tipo de dato que usaremos serán números enteros (sin decimales), que se indican con "entero". Después de esta palabra se indica el nombre que tendrá la variable:

```
entero PrimerNumero;
```

Esa orden reserva espacio para almacenar un número entero, que podrá tomar distintos valores, y al que nos referiremos con el nombre "PrimerNumero".

1.5.2. Asignación de valores

Podemos darle un valor a esa variable durante el programa haciendo

```
PrimerNumero = 234;
```

O también podemos darles un valor inicial ("inicializarlas") antes de que empiece el programa, en el mismo momento en que las definimos:

```
entero PrimerNumero = 234;
```

O incluso podemos definir e inicializar más de una variable a la vez

```
entero PrimerNumero = 234, SegundoNumero = 567;
```

(esta línea reserva espacio para dos variables, que usaremos para almacenar números enteros; una de ellas se llama PrimerNumero y tiene como valor inicial 234 y la otra se llama SegundoNumero y tiene como valor inicial 567).

Después ya podemos hacer operaciones con las variables, igual que las hacíamos con los números:

```
suma = PrimerNumero + SegundoNumero;
```

1.5.3. Constantes

Una variable de un tipo primitivo declarada como contante no puede cambiar su valor a lo largo de la ejecución del programa.

La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos. La variable definida como constante (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Podemos declarar una constante de la siguiente manera:

```
constante entero NumeroNoCambia = 123;
```

1.5.4. Mostrar el valor de una variable en pantalla

Una vez que sabemos cómo mostrar un número en pantalla, es sencillo mostrar el valor de una variable. Para un número hacíamos cosas como

```
Escribir(3+4);
```

pero si se trata de una variable es idéntico:

```
Escribir(suma);
```

O bien, si queremos mostrar un texto además del valor de la variable, podemos indicar el texto entre comillas, detallando con {0} en qué parte del texto queremos que aparezca el valor de la variable, de la siguiente forma:

```
Escribir("La suma es {0}", suma);
```

Si se trata de más de una variable, indicaremos todas ellas tras el texto, y detallaremos dónde debe aparecer cada una de ellas, usando {0}, {1} y así sucesivamente:

```
Escribir("La suma de {0} y {1} es {2}", PrimerNumero, SegundoNumero, suma);
```

Ya sabemos todo lo suficiente para crear nuestro programa que sume dos números usando variables:

```
Inicio
    entero PrimerNumero;
    entero segundoNumero; entero suma;
    PrimerNumero = 234; SegundoNumero = 567;
    suma = PrimerNumero + SegundoNumero;
    Escribir("La suma de {0} y {1} es {2}", PrimerNumero,
    SegundoNumero, suma);
Fin
```

1.6. Identificadores

Estos nombres de variable (lo que se conoce como "identificadores") pueden estar formados por letras, números o el símbolo de subrayado (_) y deben comenzar por letra o subrayado. No deben tener espacios entre medias, y hay que recordar que las vocales acentuadas y la eñe son problemáticas, porque no son letras "estándar" en todos los idiomas.

Por eso, no son nombres de variable válidos:

1numero	(empieza por número)
un numero	(contiene un espacio)
Año1	(tiene una eñe)
MásDatos	(tiene una vocal acentuada)

Tampoco podremos usar como identificadores las palabras reservadas. Por ejemplo, la palabra "entero" se refiere a que cierta variable guardará un número entero, así que esa palabra "entero" no la podremos usar tampoco como nombre de variable.

Hay que recordar que las mayúsculas y minúsculas se consideran diferentes, de modo que si intentamos hacer

```
PrimerNumero = 0; primernumero = 0;
```

o cualquier variación similar, el compilador protestará y nos dirá que no conoce esa variable, porque la habíamos declarado como:

```
entero PrimerNumero;
```

1.7. Comentarios

Podemos escribir comentarios, que el compilador ignora, pero que pueden servir para aclararnos cosas a nosotros. Se escriben entre /* y */:

```
entero suma; /* Porque guardaré el valor para usarlo más tarde */
```

Es conveniente escribir comentarios que aclaren la misión de las partes de nuestros programas que puedan resultar menos claras a simple vista. Incluso suele ser aconsejable que el programa comience con un comentario, que nos recuerde qué hace el programa sin que necesitemos mirarlo de arriba a abajo.

```
/* Ejemplo 1: Sumar dos números */
Inicio
    entero PrimerNumero;
    entero SegundoNumero;
    entero suma; /* Guardaré el valor para usarlo más tarde */
    PrimerNumero = 234; SegundoNumero = 567;
    /* Primero calculo la suma */
    suma = PrimerNumero + SegundoNumero;
    /* Y después muestro su valor */
    Escribir("La suma de {0} y {1} es {2}", PrimerNumero,
    SegundoNumero, suma);
Fin
```

Un comentario puede empezar en una línea y terminar en otra distinta, así:

```
/* Esto
    es un comentario que
    ocupa más de una línea
*/
```

También es posible declarar otro tipo de comentarios, que comienzan con doble barra y terminan cuando se acaba la línea (estos comentarios, claramente, no podrán ocupar más de una línea). Son los "comentarios de linea":

```
// Este es un comentario de línea
```

1.8. Datos por el usuario: Leer

Si queremos que sea el usuario de nuestro programa quien teclee los valores, necesitamos una nueva orden, que nos permita leer desde teclado. Pues bien, al igual que tenemos Escribir("Escribir línea"), también existe Leer(Variable). Para leer textos, haríamos:

```
Leer(texto);
```

Pero eso ocurrirá en el próximo tema, cuando veamos cómo manejar textos. De momento, nosotros sólo sabemos manipular números enteros de la siguiente forma:

```
Leer(PrimerNumero);
```

Un ejemplo de programa que sume dos números tecleados por el usuario sería:

```
Inicio
    entero PrimerNumero;
    entero SegundoNumero;
    entero suma;
    Escribir("Introduce el primer número");
    Leer(PrimerNumero);
    Escribir("Introduce el segundo número");
    Leer(SegundoNumero);
    suma = PrimerNumero + SegundoNumero;
    Escribir("La suma de {0} y {1} es {2}", PrimerNumero,
    SegundoNumero, suma);
Fin
```

2. Estructuras de control

2.1. Estructuras alternativas

2.1.1. Si

Vamos a ver cómo podemos comprobar si se cumplen condiciones. La primera construcción que usaremos será "si ... entonces ...". El formato es

Si(condición)Entonces sentencia;

Vamos a verlo con un ejemplo:

```
Inicio
    entero numero;
    Escribir("Introduce un número");
    Leer(numero);
    Si (numero>0) Entonces
        Escribir("El número es positivo.");
    FinSi
Fin
```

Este programa pide un número al usuario. Si es positivo (mayor que 0), escribe en pantalla "El número es positivo."; si es negativo o cero, no hace nada.

2.1.2. Operadores relacionales: <, <=, >, >=, ==, !=

Hemos visto que el símbolo ">" es el que se usa para comprobar si un número es mayor que otro. El símbolo de "menor que" también es sencillo, pero los demás son un poco menos evidentes, así que vamos a verlos:

Operador	Operación
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual a
!=	No igual a (distinto de)

2.1.3. Si – Sino

Podemos indicar lo que queremos que ocurra en caso de que no se cumpla la condición, usando la orden "Sino" (en caso contrario), así:

```
Inicio
    entero numero;
    Escribir("Introduce un número");
    Leer(numero);
    Si (numero>0) Entonces
        Escribir("El número es positivo.");
    Sino
        Escribir("El número es cero o negativo.");
    FinSi
Fin
```

2.1.4. Operadores lógicos: &&, ||, !

Estas condiciones se puede encadenar con "y", "o", etc., que se indican de la siguiente forma:

Operador	Significado
&&	Y
||	O
!	No

De modo que podremos escribir cosas como:

```
Si((opcion==1) && (usuario==2)) ...
Si((opcion==1) || (opcion==3)) ...
Si ((!(opcion==opcCorrecta)) ...
```

2.1.5. Según

Si queremos ver varios posibles valores, sería muy pesado tener que hacerlo con muchos "Si" seguidos o encadenados. La alternativa es la orden "Segun", cuya sintaxis es:

```
Segun (variable_numerica) Hacer
    caso 1: secuencia_de_acciones_1; romper;
    caso 2: secuencia_de_acciones_2; romper;
    caso 3: secuencia_de_acciones_3; romper;
    defecto: secuencia_acciones_por_defecto; romper;
FinSegun
```

Es decir, se escribe tras "Segun" la expresión a analizar, entre paréntesis. Después, tras varias órdenes "Caso" se indica cada uno de los valores posibles. Los pasos (porque pueden ser varios) que se deben dar si se trata de ese valor se indican a continuación, terminando con "Parar". Si hay que hacer algo en caso de que no se cumpla ninguna de las condiciones, se detalla después de las palabras "defecto".

2.2. Estructuras Repetitivas

Hemos visto cómo comprobar condiciones, pero no cómo hacer que una cierta parte de un programa se repita un cierto número de veces o mientras se cumpla una condición (lo que llamaremos un "bucle").

2.2.1. Mientras

Si queremos hacer que una sección de nuestro programa se repita mientras se cumpla una cierta condición, usaremos la orden "Mientras". Esta orden tiene dos formatos distintos, según comprobemos la condición al principio o al final.

En el primer caso, su sintaxis es:

```
Mientras (expresion_logica) Hacer
    secuencia_de_acciones
FinMientras
```

Es decir, la sentencia se repetirá mientras la condición sea cierta. Si la condición es falsa ya desde un principio, la sentencia no se ejecuta nunca.

Un ejemplo que nos diga si cada número que tecleemos es positivo o negativo, y que pare cuando tecleemos el número 0, podría ser:

```
Inicio
    entero numero;
    Escribir("Teclea un número (0 para salir): ");
    numero = ConvertirEntero(Leer());
    Mientras (numero != 0) Hacer
        Si (numero > 0) Entonces
            Escribir("Es positivo");
        Sino
            Escribir("Es negativo");
        FinSi
        Escribir("Teclea otro número (0 para salir): ");
        numero = ConvertirEntero(Leer());
    FinMientras
Fin
```

2.2.2. Repetir – Hasta Que

Este es el otro formato que puede tener la orden "Mientras": la condición se comprueba al final. El punto en que comienza a repetirse se indica con la orden "Repetir", así:

```
Repetir
    secuencia_de_acciones
Hasta Que (expresion_logica);
```

2.2.3. Para

Ésta es la orden que usaremos habitualmente para crear partes del programa que se repitan un cierto número de veces. El formato de "Para" es:

```
Para (entero i=0;i<=ValorFinal;i++) Hacer
        secuencia_de_acciones
FinPara
```

Así, para contar del 1 al 10, tendríamos 1 como valor inicial, <=10 como condición de repetición, y el incremento sería de 1 en 1. Es muy habitual usar la letra "i" como contador, cuando se trata de tareas muy sencillas, así que el valor inicial sería "i=1", la condición de repetición sería "i<=10" y el incremento sería "i=i+1":

```
Para (entero i=1;i<=10;i++) Hacer
        secuencia_de_acciones
FinPara
```

La orden para incrementar el valor de una variable ("i = i+1") se puede escribir de la forma abreviada "i++", como veremos con más detalle en el próximo tema.

2.3. Sentencia romper: termina el bucle

Podemos salir de un bucle "Para" antes de tiempo con la orden "romper":

```
Inicio
        entero contador;
        Para(contador=1; contador<=10; contador++)Hacer
                Si (contador==5) Entonces
                        romper;
                FinSi
                Escribir("{0} ", contador);
        FinPara
Fin
```

El resultado de este programa es:

```
1 2 3 4
```

2.4. Sentencia continue: fuerza la siguiente iteración

Podemos saltar alguna repetición de un bucle con la orden "continue":

```
Inicio
    entero contador;
    Para(contador=1; contador<=10; contador++)Hacer
        Si (contador==5) Entonces
            continue;
        FinSi
        Escribir("{0} ", contador);
    FinPara
Fin
```

El resultado de este programa es:

1 2 3 4 6 7 8 9 10

En él podemos observar que no aparece el valor 5.

2.5. Pausa

Podemos detener la pantalla de la consola con el comando “Pausa”. El formato de "Pausa" es:

```
Pausa();
```

3. Tipos de datos básicos

3.1. Tipo entero

Hemos hablado de números enteros, de cómo realizar operaciones sencillas y de cómo usar variables para reservar espacio y poder trabajar con datos cuyo valor no sabemos de antemano. El formato para declarar números enteros es:

```
entero numero1;
```

3.1.1. Conversiones de cadena a entero

Si queremos obtener estos datos a partir de una cadena de texto, podemos usar el comando ConvertirEntero. El formato para declarar números enteros es:

```
ConvertirEntero(“20”);
```

3.1.2. Incremento y decremento

Conocemos la forma de realizar las operaciones aritméticas más habituales. Pero también existe una operación que es muy frecuente cuando se crean programas, y que no tiene un símbolo específico para representarla en matemáticas: incrementar el valor de una variable en una unidad:

```
a = a + 1;
```

Pues bien, existe una notación más compacta para esta operación, y para la opuesta (el decremento):

a++;	es lo mismo que	a = a+1;
a--;	es lo mismo que	a = a-1;

Y ya que estamos hablando de las asignaciones, hay que comentar que es posible hacer asignaciones múltiples:

```
a = b = c = 1;
```

3.1.3. Operaciones abreviadas: +=

Pero aún hay más. Tenemos incluso formas reducidas de escribir cosas como "a = a+5". Allá van:

a += b;	es lo mismo que	a = a+b;
a -= b;	es lo mismo que	a = a-b;
a *= b;	es lo mismo que	a = a*b;
a /= b;	es lo mismo que	a = a/b;
a %= b;	es lo mismo que	a = a%b;

3.2. Tipo Real

Cuando queremos almacenar datos con decimales, no nos sirve el tipo de datos "entero". Necesitamos otro tipo de datos que sí esté preparado para guardar números "reales" (con decimales).

Para definirlos, se hace igual que en el caso de los números enteros:

```
real NumeroReal;
```

O bien, si queremos dar un valor inicial en el momento de definirlos (recordando que para las cifras decimales no debemos usar una coma, sino un punto):

```
real NumeroReal = 10.45;
```

3.3. Tipo de dato carácter

3.3.1. Leer y mostrar caracteres

También tenemos un tipo de datos que nos permite almacenar una única letra, el tipo "caracter":

```
caracter letra;
```

Asignar valores es sencillo: el valor se indica entre comillas simples

```
letra = 'a';
```

Para leer valores desde teclado, lo podemos hacer de forma similar a los casos anteriores: leemos toda una frase con el comando “Leer”:

```
Leer(letra);
```

3.3.2. Secuencias de escape: \n y otras

Como hemos visto, los textos que aparecen en pantalla se escriben con Escribir, indicados entre paréntesis y entre comillas dobles. Entonces surge una dificultad: ¿cómo escribimos una comilla doble en pantalla? La forma de conseguirlo es usando ciertos caracteres especiales, lo que se conoce como "secuencias de escape". Existen ciertos caracteres especiales que se pueden escribir después de una barra invertida (\) y que nos permiten conseguir escribir esas comillas dobles y algún otro carácter poco habitual. Por ejemplo, con \" se escribirán unas comillas dobles, y con \' unas comillas simples, o con \n se avanzarán a la línea siguiente de pantalla.

Estas secuencias especiales son las siguientes:

Secuencia	Significado
\a	Emite un pitido
\b	Retroceso (permite borrar el último carácter)
\f	Avance de página (expulsa una hoja en la impresora)
\n	Avanza de línea (salta a la línea siguiente)
\r	Retorno de carro (va al principio de la línea)

\t	Salto de tabulación horizontal
\v	Salto de tabulación vertical
\'	Muestra una comilla simple
\"	Muestra una comilla doble
\\	Muestra una barra invertida
\0	Carácter nulo

3.4. Toma de contacto con las cadenas de texto

Las cadenas de texto son tan fáciles de manejar como los demás tipos de datos que hemos visto, con apenas tres diferencias:

- Se declaran con "cadena".
- Si queremos dar un valor inicial, éste se indica entre comillas dobles.
- Cuando leemos con Leer, no hace falta convertir el valor obtenido.
- Podemos comparar su valor usando "==" o "!=".

3.5. Los valores "booleanos"

Tenemos también un tipo de datos llamado "booleano", que puede tomar dos valores: verdadero o falso:

```
booleano encontrado;
encontrado = verdadero;
```

Este tipo de datos hará que podamos escribir de forma sencilla algunas condiciones que podrían resultar complejas.

4. Arrays, estructuras y cadenas de texto

4.1. Conceptos básicos sobre arrays

4.1.1. Definición de un array y acceso a los datos

Una vector, matriz o array (que algunos autores traducen por "arreglo") es un conjunto de elementos, todos los cuales son del mismo tipo. Estos elementos tendrán todo el mismo nombre, y ocuparán un espacio contiguo en la memoria.

Por ejemplo, si queremos definir un grupo de números enteros, el tipo de datos que usaremos para declararlo será "entero[]". Si sabemos desde el principio cuantos datos tenemos (por ejemplo 4), les reservaremos espacio con "= nuevo entero[4]", así

```
entero[] ejemplo = nuevo entero[4];
```

Podemos acceder a cada uno de los valores individuales indicando su nombre (ejemplo) y el número de elemento que nos interesa, pero con una precaución: se empieza a numerar desde 0, así que en el caso anterior tendríamos 4 elementos, que serían ejemplo[0], ejemplo[1], ejemplo[2], ejemplo[3].

Como ejemplo, vamos a definir un grupo de 5 números enteros y hallar su suma:

```
Inicio
    entero[] numero = nuevo entero[5];
    entero suma;
    numero[0] = 200;
    numero[1] = 150;
    numero[2] = 100;
    numero[3] = -50;
    numero[4] = 300;
    suma = numero[0] + numero[1] + numero[2] +
    numero[3] + numero[4];
    Escribir("Su suma es {0}", suma);
Fin
```

4.1.2. Valor inicial de un array

Al igual que ocurría con las variables "normales", podemos dar valor a los elementos de una tabla al principio del programa. Será más cómodo que dar los valores uno por uno, como hemos hecho antes, pero sólo se podrá hacer si conocemos todos los valores. En este caso, los indicaremos todos entre llaves, separados por comas:

```
Inicio
        entero[] numero = {200,150,100,-50,300};
        entero suma;
        suma = numero[0] + numero[1] + numero[2] +
        numero[3] + numero[4];
        Escribir("Su suma es {0}", suma);
Fin
```

4.1.3. Recorriendo los elementos de una tabla

Es de esperar que exista una forma más cómoda de acceder a varios elementos de un array, sin tener siempre que repetirlos todos, como hemos hecho en

suma = numero[0] + numero[1] + numero[2] + numero[3] + numero[4];

El "truco" consistirá en emplear cualquiera de las estructuras repetitivas que ya hemos visto (Mientras, Repetir...Hasta Que, Para), por ejemplo así:

```
suma = 0;
Para (i=0; i<=4; i++) Hacer
        suma += numero[i];
FinPara
```

El código fuente completo podría ser así:

```
Inicio
        entero[] numero = {200,150,100,-50,300};
        entero suma;
        Para (i=0; i<=4; i++) Hacer
                suma += numero[i];
        FinPara
        Escribir("Su suma es {0}", suma);
Fin
```

En este caso, que sólo sumábamos 5 números, no hemos escrito mucho menos, pero si trabajásemos con 100, 500 o 1000 números, la ganancia en comodidad sí que está clara.

4.1.4. Datos repetitivos introducidos por el usuario

Si queremos que sea el usuario el que introduzca datos a un array, usaríamos otra estructura repetitiva ("Para", por ejemplo) para pedírselos:

```
Inicio
        entero[] numero =  nuevo entero[5];
        entero suma;
        Para (i=0; i<=4; i++) Hacer
                Escribir("Introduce el dato numero {0}: ", i+1);
                entero n1;
                Leer(n1);
                numero[i]= n1;

        FinPara
        Para (i=0; i<=4; i++) Hacer
                suma += numero[i];
        FinPara
        Escribir("Su suma es {0}", suma);
Fin
```

4.2. Tablas bidimensionales

Podemos declarar tablas de dos o más dimensiones. Por ejemplo, si queremos guardar datos de dos grupos de alumnos, cada uno de los cuales tiene 20 alumnos, tenemos dos opciones:

- Podemos usar entero datosAlumnos[40] y entonces debemos recordar que los 20 primeros datos corresponden realmente a un grupo de alumnos y los 20 siguientes a otro grupo. Es "demasiado artesanal", así que no daremos más detalles.

- bien podemos emplear entero datosAlumnos[2,20] y entonces sabemos que los datos de la forma datosAlumnos[0,i] son los del primer grupo, y los datosAlumnos[1,i] son los del segundo.

En cualquier caso, si queremos indicar valores iniciales, lo haremos entre llaves, igual que si fuera una tabla de una única dimensión.

Vamos a ver un primer ejemplo del uso con arrays de la forma [2,20], lo que podríamos llamar el "estilo Pascal", en el usemos tanto arrays con valores prefijados, como arrays para los que reservemos espacio con "nuevo" y a los que demos valores más tarde:

```
Inicio
    entero[,] notas1 = nuevo entero[2,2];
    notas1[0,0] = 1;
    notas1[0,1] = 2;
    notas1[1,0] = 3;
    notas1[1,1] = 4;
    entero[,] notas2 = { {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {11, 12, 13,
    14, 15, 16, 17, 18, 19, 20} };
    Escribir("La nota1 del segundo alumno del grupo 1 es {0}",
    notas1[0,1]);
    Escribir("La nota2 del tercer alumno del grupo 1 es {0}",
    notas2[0,2]);
Fin
```

Este tipo de tablas de varias dimensiones son las que se usan también para guardar matrices, cuando se trata de resolver problemas matemáticos más complejos que los que hemos visto hasta ahora.

La otra forma de tener arrays multidimensionales son los "arrays de arrays", que, como ya hemos comentado, y como veremos en este ejemplo, pueden tener elementos de distinto tamaño. En ese caso nos puede interesar saber su longitud, para lo que podemos usar "Longitud":

```
Inicio
    entero[][] notas;
    notas = nuevo entero[3][];
    notas[0] = nuevo entero[10];
    notas[1] = nuevo entero[15];
    notas[2] = nuevo entero[12];

    Para(entero i=0;i< Longitud(notas);i++)Hacer
        Para (entero j=0;j< Longitud(notas[i]);j++) Hacer
            notas[i][j] = i + j;
        FinPara
    FinPara
    Para(entero i=0;i< Longitud(notas);i++) Hacer
        Para (entero j=0;j< Longitud(notas[i]);j++) Hacer
            Escribir(" {0}", notas[i][j]);
        FinPara
    FinPara
    Pausa();
Fin
```

4.3. Cadenas de caracteres

4.3.1. Definición. Lectura desde teclado

Hemos visto cómo leer cadenas de caracteres (Leer) y cómo mostrarlas en pantalla (Escribir), así como la forma de darles un valor(=). También podemos comparar cuál es su valor, usando ==, o formar una cadena a partir de otras si las unimos con el símbolo de la suma (+):

Así, un ejemplo que nos pidiese nuestro nombre y nos saludase usando todas estas posibilidades podría ser:

```
Inicio
        cadena saludo = "Hola";
        cadena segundoSaludo;
        cadena nombre, despedida;
        segundoSaludo = "Que tal?";
        Escribir("Dime tu nombre... ");
        Leer(nombre);
        Escribir("{0} {1}",saludo,nombre);
        Escribir(segundoSaludo);
        Si(nombre == "Alberto") Entonces
                Escribir("Dices que eres Alberto?");
        Sino
                Escribir("Así que no eres Alberto?");
        FinSi
        despedida = "Adios " + nombre + "!";
        Escribir(despedida);
Fin
```

4.3.2. Cómo acceder a las letras que forman una cadena

Podemos leer una de las letras de una cadena, de igual forma que leemos los elementos de cualquier array: si la cadena se llama "texto", el primer elemento será texto[0], el segundo será texto[1] y así sucesivamente.

Eso sí, las cadenas no se pueden modificar letra a letra: no podemos hacer texto[0]='a'. Para eso habrá que usar una construcción auxiliar, que veremos más adelante.

4.3.3. Longitud de la cadena

Podemos saber cuántas letras forman una cadena con la función "Longitud". Esto permite que podamos recorrer la cadena letra por letra, usando construcciones como "Para".

4.3.4. Extraer una subcadena

Podemos extraer parte del contenido de una cadena con "SubCadena", que recibe dos parámetros: la posición a partir de la que queremos empezar y la cantidad de caracteres que queremos obtener. El resultado será otra cadena:

saludo = SubCadena("Hola mundo",0,4);

4.3.5. Concatenar una cadena

Podemos concadenar 2 o más cadenas y obtener una nueva cadena. El resultado será otra cadena:

NombresApellidos = Concatenar(“Juan”,”Solis”);

4.3.6. Otras manipulaciones de cadenas

Ya hemos comentado que las cadenas son inmutables, no se pueden modificar. Pero sí podemos realizar ciertas operaciones sobre ellas para obtener una nueva cadena. Por ejemplo:

- Mayuscula(CadenaTexto) convierte a mayúsculas: Mayuscula("hola mundo");
- Minuscula(CadenaTexto) convierte a minúsculas una cadena especifica: Minuscula("HOLA MUNDO");
- Insertar(Cadena, CadenaInsertar, Posición): Insertar una subcadena en una cierta posición de la cadena inicial: Insertar("Hola Mundo","2015",9);
- Remover(CadenaTexto, Posición): Elimina una cantidad de caracteres en cierta posición: Remover("Hola Mundo" ,5);
- Remplazar(CadenaTexto, CadenaBusqueda, CadenaRemplazar): Sustituye una cadena (todas las veces que aparezca) por otra: Remplazar ("Hola Juan", "Juan","Ana");

4.3.7. Comparación de cadenas

Sabemos comprobar si una cadena tiene exactamente un cierto valor, con el operador de igualdad (==). En su lugar, debemos usar "Comparar", que devolverá un valor booleano verdadero si las cadenas son igual y falso en caso no sean iguales:

```
Si(Comparar("hola","mundo") == falso)
        Escribir("Las cadenas no son iguales");
```

También podemos comparar sin distinguir entre mayúsculas y minúsculas, usando Comparar, al que indicamos las dos cadenas y un tercer dato "verdadero" cuando queramos ignorar esa distinción:

```
Si(Comparar("hola", "HOLA", verdadero) ==verdadero)
        Escribir ("Las cadenas son iguales");
```

5. Funciones y procedimientos

5.1. Diseño modular de programas: Descomposición modular

Hasta ahora hemos estado pensando los pasos que deberíamos dar para resolver un cierto problema, y hemos creado programas a partir de cada uno de esos pasos. Esto es razonable cuando los problemas son sencillos, pero puede no ser la mejor forma de actuar cuando se trata de algo más complicado.

- A partir de ahora vamos a empezar a intentar descomponer los problemas en trozos más pequeños, que sean más fáciles de resolver. Esto nos puede suponer varias ventajas:
- Cada "trozo de programa" independiente será más fácil de programar, al realizar una función breve y concreta.
- El "programa principal" será más fácil de leer, porque no necesitará contener todos los detalles de cómo se hace cada cosa.
- Podremos repartir el trabajo, para que cada persona se encargue de realizar un "trozo de programa", y finalmente se integrará el trabajo individual de cada persona.

Esos "trozos" de programa son lo que suele llamar "subrutinas", "procedimientos" o "funciones".

5.2. Funciones

Una función, desde el punto de vista de la programación, se define como un proceso que recibe valores de entrada (llamados parámetros) y el cual retorna un valor resultado. Adicionalmente, las funciones son subprogramas dentro de un programa, que se pueden invocar (ejecutar) desde cualquier parte del programa, es decir, desde otra función, desde la misma función o desde el programa principal, cuantas veces sea necesario.

Las funciones se usan cuando existen dos o más porciones de algoritmo dentro de un programa que son iguales o muy similares, por ejemplo, en un algoritmo se puede emplear varias veces una porción de algoritmo que eleva a una potencia dada un número real.

De esta manera conviene definir una función que al ser invocada ejecute dicho código, y en el lugar donde estaba la porción de algoritmo original, se hace un llamado (ejecución) de la función creada.

En el pseudocódigo una función se declara de la siguiente manera:

```
Funcion cadena Saludar()
        cadena mensaje="Bienvenido al programa";
                mensaje= mensaje + " de ejemplo.";
                mensaje= mensaje + " Espero que estés bien";
FinFuncion
```

Todos los "trozos de programa" son funciones, incluyendo el propio cuerpo de programa. De hecho, la forma básica de definir una función será indicando su nombre seguido de unos paréntesis vacíos, como hacíamos con "Proceso Principal. Después, indicaremos todos los pasos que queremos que dé ese "trozo de programa".

```
Funcion cadena Saludar()
        cadena mensaje="Bienvenido al programa";
                mensaje= mensaje + " de ejemplo.";
                mensaje= mensaje + " Espero que estés bien";
FinFuncion
```

Ahora desde dentro del cuerpo de nuestro programa, podríamos "llamar" a esa función:

```
Inicio
        cadena mensaje = saludar();
        Escribir("Mensaje saluda es:" + mensaje);
Fin
```

Así conseguimos que nuestro programa principal sea más fácil de leer.

Un detalle importante: tanto la función habitual "Proceso Principal" como la nueva función "Saludar" serían parte del mismo pseudocódigo, es decir, la fuente completa sería así:

```
Funcion cadena Saludar()
        cadena mensaje="Bienvenido al programa";
                mensaje= mensaje + " de ejemplo.";
                mensaje= mensaje + " Espero que estés bien";
FinFuncion
Inicio
        cadena mensaje = saludar();
        Escribir("Mensaje saluda es:" + mensaje);
Fin
```

5.2.1. Parámetros de una función

Es muy frecuente que nos interese además indicarle a nuestra función ciertos datos especiales con los que queremos que trabaje. Por ejemplo, si escribimos en pantalla la suma de dos numeros. Lo podríamos hacer así:

```
Funcion  entero Suma( entero n1, entero n2 )
        entero suma = n1+n2;
        retornar suma;
FinFuncion
```

(La función Suma retorna un numero entero como resultado de la suma de los dos numeros).

```
Funcion  entero Suma( entero n1, entero n2 )
        entero suma = n1+n2;
        retornar suma;
FinFuncion


Inicio
        Entero numero1, numero2, respuesta;
        numero1= 4;
        numero2= 5;
        respuesta=numero1+numero2;
        Escribir("La suma de los números es: "+ respuesta);
        Pausa();
Fin
```

Estos datos adicionales que indicamos a la función es lo que llamaremos sus "parámetros". Como se ve en el ejemplo, tenemos que indicar un nombre para cada parámetro (puede haber varios) y el tipo de datos que corresponde a ese parámetro. Si hay más de un parámetro, deberemos indicar el tipo y el nombre para cada uno de ellos, y separarlos entre comas:

```
Funcion entero Resta (entero x, entero y, entero z)
        ...
FinFuncion
```

5.2.2. Valor devuelto por una función.

Las funciones matemáticas que estamos acostumbrados a manejar: sí devuelven un valor, el resultado de una operación.

De igual modo, para nosotros también será habitual que queramos que nuestra función realice una serie de cálculos y nos "devuelva" el resultado de esos cálculos, para poderlo usar desde cualquier otra parte de nuestro programa.

Por ejemplo, podríamos crear una función para elevar un número entero al cuadrado así:

```
Funcion entero cuadrado (entero n)
        retornar n*n;
FinFuncion
```

Y podríamos usar el resultado de esa función como si se tratara de un número o de una variable, así:

```
resultado = cuadrado( 5 );
```

Un programa más detallado de ejemplo podría ser:

```
Funcion entero cuadrado ( entero n )
        retornar n*n;
FinFuncion
Inicio
        entero numero;
        entero resultado;
        numero= 5;
        resultado = cuadrado(numero);
        Escribir("El cuadrado del numero {0} es {1}",
        numero, resultado);
        Escribir(" y el de 3 es {0}", cuadrado(3));
Fin
```

5.2.3. Funciones recursivas

Una función recursiva es una función que se define en términos de si misma, es decir, que el resultado de la función depende de resultados obtenidos de evaluar la misma función con otros valores.

Se debe tener mucho cuidado en la definición de funciones recursivas, pues si no se hace bien, la función podría requerir de un cálculo infinito o no ser calculable.

Uno clásico es el "factorial de un número":

$$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1$$

(por ejemplo, el factorial de 4 es $4 \cdot 3 \cdot 2 \cdot 1 = 24$)

Si pensamos que el factorial de n-1 es

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdot ... \cdot 3 \cdot 2 \cdot 1$$

Entonces podemos escribir el factorial de un número a partir del factorial del siguiente número:

$$n! = n \cdot (n-1)!$$

Esta es la definición recursiva del factorial, ni más ni menos. Esto, programando, se haría:

```
Inicio
        entero factorial, numero;
        Escribir("Ingrese un numero:");
        Leer(numero);
        factorial= Factorial(numero);
        Escribir("El factorial del numero es:"+factorial);
        Pausa();
Fin
Funcion entero Factorial(entero n)
Si(n==0)Entonces
        retornar 1;
Sino
        retornar n * Factorial(n-1);
FinSi
FinFuncion
```

Dos consideraciones importantes:

- Atención a la primera parte de la función recursiva: es MUY IMPORTANTE comprobar que hay salida de la función, para que nuestro programa no se quede dando vueltas todo el tiempo y deje el ordenador (o la tarea actual) "colgado".
- Los factoriales crecen rápidamente, así que no conviene poner números grandes: el factorial de 16 es 2.004.189.184, luego a partir de 17 podemos obtener resultados erróneos, si usamos números enteros "normales".

5.3. Algunas funciones útiles

5.3.1. Números aleatorios

En un programa de gestión o una utilidad que nos ayuda a administrar un sistema, no es habitual que podamos permitir que las cosas ocurran al azar. Pero los juegos se encuentran muchas veces entre los ejercicios de programación más completos, y para un juego sí suele ser conveniente que haya algo de azar, para que una partida no sea exactamente igual a la anterior.

Generar números al azar ("números aleatorios") usando no es difícil: debemos utilizar la función “Aleatorio", para obtener valores entre dos extremos:

```
entero aleatorio = Aleatorio(1, 101);
```

Vamos a ver un ejemplo, que muestre en pantalla dos números al azar entre 1 y 10:

```
Inicio
    entero aleatorio = Aleatorio (1, 11);
    Escribir("Un número entre 1 y 10: {0}", aleatorio);
    entero aleatorio2 = Aleatorio (1, 11);
    Escribir("Otro: {0}", aleatorio2);
Fin
```

5.3.2. Funciones matemáticas

Tenemos muchas funciones matemáticas predefinidas, como:

- ABS(x): Valor absoluto
- Acos(x): Arco coseno
- Asin(x): Arco seno
- Atan(x): Arco tangente
- Coseno(x): Coseno
- Exponencial(x): Exponencial de x (e elevado a x)
- Logaritmo(x): Logaritmo natural (o neperiano, en base "e")
- Logaritmo10(x): Logaritmo en base 10
- Potencia(x,y): x elevado a y
- Redondeo(x, cifras): Redondea un número
- Seno(x): Seno
- SQRT(x): Raíz cuadrada
- Tan(x): Tangente

(casi todos ellos usan parámetros X e Y de tipo "real") y una serie de constantes como E, el número "e", con un valor de 2.71828... PI, el número "Pi", 3.14159...

La mayoría de ellas son específicas para ciertos problemas matemáticos, especialmente si interviene la trigonometría o si hay que usar logaritmos o exponenciales. Pero vamos a destacar las que sí pueden resultar útiles en situaciones más variadas:

La raíz cuadrada de 4 se calcularía haciendo x = SQRT(4);

La potencia: para elevar 2 al cubo haríamos y = Potencia(2, 3);

El valor absoluto: para trabajar sólo con números positivos n = ABS(x);

5.4. Procedimientos

En muchos casos existen porciones de código similares que no calculan un valor si no que por ejemplo, presentan información al usuario, leen una colección de datos o calculan más de un valor. Como una función debe retornar un único valor este tipo de porciones de código no se podrían codificar como funciones. Para superar este inconveniente se creó el concepto de procedimiento.

Un procedimiento se puede asimilar a una función que puede retornar más de un valor mediante el uso de parámetros por referencia. Los procedimientos se usan para evitar duplicación de código y conseguir programas más cortos.

Son también una herramienta conceptual para dividir un problema en subproblemas logrando de esta forma escribir más fácilmente programas grandes y complejos.

```
Inicio
        Mensaje("Mensaje con procedimeinto");
Fin
Procedimiento  Mensaje(cadena m)
        Escribir("Mensaje es:"+m);
        Pausa();
FinProcedimiento
```

También podemos hacer uso de valores de referencia para devolver más de un valor:

```
Inicio
        entero numero1, numero2, suma, resta;
        numero1=2;
        numero2=3;
        suma=0;
        resta=0;
        Operaciones(numero1,numero2,referencia suma,referencia
        resta);
        Escribir("La suma es:"+suma+"\n");
```

```
        Escribir("La resta es:"+resta);
        Pausa();
    Fin
    Procedimiento  Operaciones(entero n1, entero n2, referencia entero sum, referencia entero rest)
        sum=n1+n2;
        rest=n1-n2;
    FinProcedimiento
```

6. Gráficos

6.1. Dibujar Punto

Dibuja un punto especificado por los pares de coordenadas X y Y

```
DibujarPunto(x1, y1, color);
```

6.2. Dibujar Línea

Dibuja una línea que conecta los dos puntos especificados por los pares de coordenadas.

```
DibujarLinea(x1, y1, x2, y2, color, ancho)
```

6.3. Dibujar Imagen

Dibuja la imagen especificada en la ubicación que se indique y con el tamaño especifico.

```
DibujarImagen(ArchivoImagen, x1, y1, ancho, altura);
```

6.4. Dibujar Texto

Dibuja la cadena de texto especificada en la ubicación especificada y con el tamaño y color especificado.

```
DibujarTexto(texto, x1, y1, tamano, color);
```

6.5. Dibujar Rectángulo

Dibuja un rectángulo especificado por un par de coordenadas, un valor de ancho y un valor de alto.

```
DibujarRectangulo(x1,y1,ancho,altura,color, grosor);
```

6.6. Dibujar Arco

Dibuja un arco especificado por un par de coordenadas, ángulo de inicio y fin específico.

DibujarArco(x1, y1, ancho, altura, InicioAngulo, FinAngulo,color, grosor);

6.7. Tiempo

Permite tener un contador de tiempo en milisegundos.

Tiempo(MiliSegundos);

6.8. Colores

Numero de colores disponibles para las gráficas.

Color	Numero
Negro	0
Blanco	1
Rojo	2
Azul	3
Verde	4
Amarillo	5
Otros	0