

**TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE VERACRUZ**

MATERIA: Lenguajes y Autómatas II

EQUIPO

Calderón Torres Irving

Daniel Luna

Jared Aramis Lucho Saunier

DOCENTE: M.E. Ofelia Gutiérrez Giraldi

PROYECTO: Compilador



Índice

Índice.....	1
Introducción.....	3
Manual de sistema.....	4
Nombre del lenguaje.....	5
Objetivo.....	5
Análisis Léxico.....	5
Token.....	5
Lexema.....	6
Análisis sintáctico.....	6
Requisitos.....	7
Tablas de símbolos.....	7
Estructura Main.....	7
Símbolos de agrupación.....	7
Operadores aritméticos.....	7
Operadores de comparación.....	8
Operadores lógicos.....	8
Sentencias.....	8
Declaración de variables.....	8
Gramática.....	10
Declaraciones:.....	10
1. Asignación:.....	10
2. Instancia:.....	10
Gramática de tokens.....	11
Token Romper.....	14
Token Número.....	14
Token String.....	14
Token Boolean.....	14
Token IF.....	14
Token Ciclo.....	15
Token Leer.....	15
Token Imprimir.....	15
Condicionales.....	16
Ciclos.....	17
Método de entrada.....	17
Método de salida.....	17
Tabla de errores.....	18
Tabla de errores por asignación de signos.....	18
Tabla de errores por punto y coma.....	18
Tabla de errores de identificador y palabras reservadas para tipo de datos.....	19
Tabla de errores en la condición Si.....	19
Tabla de errores del ciclo mientras.....	19
Tabla de errores del ciclo para.....	20
Tabla de errores para el método leer.....	20
Tabla de errores para el método imprimir.....	20

Sintaxis de sentencia de lenguaje.....	22
Análisis semántico.....	23

Introducción

En el presente documento se dará presentación al lenguaje de programación realizado a lo largo del semestre de Autómatas II, referenciando a nuestro compilador.

Para comenzar a trabajar en nuestro proyecto nuestra computadora hace uso de una herramienta denominada “Compilador” que funciona para traducir nuestro lenguaje de programación.

En este caso, vamos a hablar sobre el compilador Epsilon y la estructura que hemos desarrollado para construirlo.

Para formular el lenguaje de programación de Epsilon, se emplearon tres métodos principales: las expresiones regulares, las gramáticas y otros procesos. Estos métodos fueron la base inicial que permitieron avanzar hacia las siguientes fases del proceso y finalmente lograr la generación de código de objeto.

Manual de sistema

Nombre del lenguaje

El lenguaje Epsilon cuenta con una sintaxis fácil y comprensible para el usuario, está principalmente dirigido para personas principiantes en programación.

Objetivo

El objetivo principal de Epsilon es ofrecer una herramienta de programación que sea fácil de manejar y entender, especialmente para aquellos que están comenzando en este campo. En este documento, se van a detallar todas las fases que se utilizan en Epsilon para llevar a cabo la tarea de programación. Estas fases incluyen la fase

léxica, sintáctica, semántica, generación de código intermedio y generación de código objeto.

Análisis Léxico

El análisis léxico, también conocido como escaneo léxico, es el proceso de análisis de un programa de ordenador para identificar y separar las diferentes partes que lo componen en unidades llamadas "lexemas".

Estas unidades pueden ser identificadores, números, operadores, palabras clave, signos de puntuación, etc. El análisis léxico es la primera fase del proceso de compilación de un programa de ordenador y es esencial para comprender el significado y la estructura del código fuente.

El analizador léxico produce como salida un token por cada lexema el cual es de la siguiente forma:

(nombre - token, valor - atributo)

En resumen, el análisis léxico se encarga de identificar los diferentes elementos léxicos que componen un programa y asignarles un significado dentro del lenguaje de programación utilizado.

Token

Un token es una unidad básica de un lenguaje de programación que representa un elemento léxico concreto. Los tokens se generan durante el análisis léxico del código fuente y son utilizados por el compilador o intérprete para construir la estructura sintáctica del programa. Un token se compone de dos partes: un tipo y un valor. El tipo de un token describe el tipo de elemento léxico que representa, como una palabra clave, un identificador, un operador, un número, etc. El valor del token es el contenido específico que se encuentra dentro del código fuente, como el nombre de una variable, el valor de un número, el operador aritmético, etc. Los tokens son utilizados por el compilador o intérprete para comprender la estructura y el significado del código fuente y así poder generar código ejecutable o ejecutar el programa.

Lexema

Un lexema es la unidad más pequeña e indivisible de un lenguaje de programación que tiene un significado propio dentro del lenguaje. En otras palabras, un lexema es una secuencia de caracteres en el código fuente que representa un elemento léxico concreto, como una palabra clave, un identificador, un literal, un operador, un signo de puntuación, etc. Durante el análisis léxico del código fuente, se identifican y separan los lexemas en tokens, que luego son utilizados por el compilador o intérprete para construir la estructura sintáctica del programa. En resumen, un lexema es una unidad fundamental del lenguaje de programación que ayuda a representar el significado de los elementos léxicos del código fuente.

Análisis sintáctico

El análisis sintáctico, es el proceso en el cual se analiza la estructura gramatical de un programa de computadora para determinar si cumple con las reglas sintácticas del lenguaje de programación utilizado. En otras palabras, el análisis sintáctico verifica si las construcciones del código fuente, como las expresiones, declaraciones y sentencias, están escritas en un orden y estructura coherente y lógica según las reglas del lenguaje. El objetivo del análisis sintáctico es crear un árbol sintáctico, también conocido como árbol de análisis, que representa la estructura gramatical del programa y permite al compilador o intérprete comprender el significado y la intención del código fuente. Si el código fuente no cumple con las reglas sintácticas del lenguaje, se generará un error sintáctico que debe ser corregido antes de que el programa pueda ser compilado o interpretado. En resumen, el análisis sintáctico es una fase importante en la compilación de un programa de computadora que verifica si la estructura del código fuente cumple con las reglas sintácticas del lenguaje de programación utilizado

Requisitos

Para que el compilador tenga su correcto funcionamiento es necesario tener los siguientes requisitos mínimos de hardware y software.

Requisitos de software:

- Windows 7.

Requisitos de hardware:

- RAM:150MB
- Espacio en disco: 5MB

Tablas de símbolos

Estructura Main

Tipo	Valor
INICIO	Inicio
FIN	Fin

Símbolos de agrupación

Tipo	Valor
LI	{
LD	}
ASOCIATIVO	:
PC	;

Operadores aritméticos

Tipo	Valor
MAS	+
MENOS	-
MULT	*
DIV	/
RESIDUO	%

Operadores de comparación

Tipo	Valor
ASIGNACION	=
IGUAL	==
MAYORQ	>
MENORQ	<
DIFERENTE	!=
NOT	!

Operadores lógicos

Tipo	Valor
OR	
AND	&

Sentencias

Tipo	Valor
IF	Si
FOR	Para
WHILE	Mientras
LEER	Leer
IMPRIMIR	Imprimir
ROMPER	Romper

Declaración de variables

Tipo	Valor
NUMERO	Num
STRING	Cad
VF	Verdadero Falso
V_NUM	(["0"-“9”]) ((["0"-“9”])+ (“.”)(["0"-“9”]))
V_ALF_NUM	(“\”)+((“ ”) (["a"-“z”]) (["A"-“Z”]) (["0"-“9”]))+ (“\ ”)
IDENTIFICADOR	((["A"-“Z”])(["0"-“9”"])*)

Palabras reservadas	Descripción
Num	Valor numérico de tipo o valor decimal
Cad	Cadenas de caracteres
Bool	Dato lógico que solo puede tener los valores verdadero o falso
Mod	Palabra reservada para obtener el residuo
+	Operador aritmético de suma
-	Operador aritmético de resta
*	Operador aritmético de multiplicación
/	Operador aritmético de división
~	Operador que nos permite concatenar cadenas o un valor
%	Operador aritmético de residuo
=	Operador que indica una asignación
==	Operador lógico que indica si existe igualdad entre dos valores
:	Operador que indica la finalización de una instrucción
<	Operador lógico menor que
>	Operador lógico mayor que
!	Operación lógica de negación
!=	Operador lógico que indica si existe diferencia entre dos valores
&	AND lógico
	OR lógico
{	Signo que hace referencia al inicio de un bloque de código
}	Signo que hace referencia al final de un bloque de código
' '	Indica que lo que vaya dentro será una cadena alfanumérica
0,1,2,...	Los números reales constantes
Inicio	Palabra que hace referencia al bloque de código donde iniciara la compilación
Fin	La compilación termina hasta encontrar esta palabra
Romper	Instrucción que da fin a un ciclo o condición
Leer	Instrucción para realizar una entrada
Imprimir	Instrucción para realizar una salida
Para	Instrucción que repetirá el bloque de código dentro de esa sección
Mientras	Instrucción que repetirá el bloque de código dentro de esa sección
Si	Condición
Verdadero	Respuesta perteneciente a la palabra
Falso	Respuesta perteneciente a la palabra

Gramática

Podemos dividir la gramática en diferentes bloques o estructuras que corresponden a distintos elementos del lenguaje de programación. Por ejemplo, podemos encontrar estructuras para declaraciones, sentencias condicionales, bucles, entrada de datos por teclado y salida por pantalla. De esta manera, la gramática se organiza en categorías que permiten una comprensión más clara y estructurada del lenguaje de programación.

Declaraciones:

Las declaraciones pueden manejarse de dos maneras, como una asignación o como una instancia.

1. Asignación:

“Identificador” “Signo de asignación” “Valor” “Punto y coma” Ejemplo: $P = 12;$

2. Instancia:

“Tipo de dato” “Identificador” “Punto y coma” Ejemplo: $\text{Num } G;$

También se permite aplicar ambas acciones.

“Tipo de dato” “Identificador” “Signo de asignación” “Valor” “Punto y coma” Ejemplo: $\text{Num } S = 0;$

La declaración de una variable es por medio una letra de la A-Z en mayúscula, así como también se le puede agregar un número del 0-9 en caso de que las quiera identificar con un número

Ejemplo: $\text{Num } A1 = 3;$

Nota: La declaración de una variable no puede comenzar con un número

Gramática de tokens

Se hace la declaración en este caso con el nombre Epsilon, generando la clase Epsilon donde está ubicado el main del compilador, podemos observar que tenemos declarado un String que forma parte de los errores.

```
3 public class Epsilon implements EpsilonConstants {
4
5     String A="\u005c-----Errores:\u005c"; //Variable para almacenar los errores
6     String[] TE = {"\u005cError L\u005c", "\u005cError Sint\u005c", "\u005cError Sem\u005c"}; //Tipos de errores
7
8     //Inicio de Variables para Sem\u005c*****
9     int ValorN;
10    Cache Objeto Variable = new Cache(); //Variable Fluctuante
11    String Type="", Valor=""; //Variables Cache para capturar el PAR
12    String lugar = ""; //Posición Variable donde se puede originar una inconsistencia
13    java.util.ArrayList<ParOrd> ParOrdenado = new java.util.ArrayList<ParOrd>(); //Lista de IDs
14    //Fin de Variables para Sem\u005c*****
15    //Inicio de Variables para C\u005c Intermedio*****
16    String ruta = "C:/Epsilon/codigointermedio.txt";
17    CTDOA codigo aritmetico = new CTDOA();
18    EscrituraCI ECI = new EscrituraCI(ruta); //Escritura de C\u005c Intermedio
19    String CI = ""; //Almac\u005c de C\u005c Intermedio
20    String IGCI = ""; //IDENTIFICADOR en uso para la GENERACIÓN de C\u005c INTERMEDIO
21    int label = 0; //Etiquetas usadas en mientras
22    boolean aplicaGI = true; //Valida que la instrucción este correcta para generar el C\u005c Intermedio
23    boolean vigilanteid = true;
24    //Fin de Variables para C\u005c Intermedio*****
25
26    //Inicio de Variables para Optimización*****
27    boolean T = false; //Debo cambiarlo. Se usa en Ingreso ID's
28    java.util.ArrayList<ParOrd> IDsinUso = new java.util.ArrayList<ParOrd>(); //lista con las variables muertas
29    //Fin de Variables para Optimización*****
30
31    //Inicio variables para Ensamblador*****
32    String Call = "cmd /k start C:/Epsilon/Ensamble.bat";
33    W_CPlusPlus trans = new W_CPlusPlus("C:/Epsilon/Implementacion.cpp");
34    //Fin variables para Ensamblador*****
35
36    public static void main(String[] args) throws ParseException{
37
38        Epsilon compilador = new Epsilon(System.in);
```

Iniciamos definiendo las palabras reservadas en la sección léxica del compilador. Identificamos los Tokens que forman parte de nuestro lenguaje y los declaramos mediante la definición del tipo TOKEN. También se establecieron los caracteres que serán filtrados por el analizador léxico, incluyendo espacios y saltos de línea.

```
TOKEN:
{
    <INICIO: "Inicio">
    |<ROMPER: "romper">
    |<LEER: "leer">
    |<IMPRIMIR: "imprimir">
    |<WHILE: "mientras">
    |<FOR: "para">
    |<V_ALF_NUM: ("\'')+((\' ")|(["a"- "z"])|(["A"- "Z"])|(["0"- "9"])|(":")+("\'")>
    |<IDENTIFICADOR: (((["A"- "Z"])(["0"- "9"])*)+>
    |<NUMERO: "Num">
    |<V_NUM: (((["0"- "9"])+) | (((["0"- "9"])+)(["."])(["0"- "9"])+)>
    |<STRING: "Cad">
    |<BOOLEAN: "Bool">
    |<VF: "Verdadero" | "Falso">
    |<IF: "si">
    |<FIN: "Fin">
    |<SWITCH: "segun">
    |<CASE: "caso">
```

```
TOKEN:
{
  <PI: "(">
  |<PD: ")">
  |<LI: "{">
  |<LD: "}">
  |<PC: ";">
  |<ASOCIATIVO: ":">
  |<IGUAL: "==">
  |<ASIGNACION: "=">
  |<COMA: ",">
  |<OR: "|">
  |<AND: "&">
  |<NOT: "!">
  |<DIFERENTE: "!=">
  |<MAYORQ: ">">
  |<MENORQ: "<">
  |<RESIDUO: "%">
  |<MODULO: "Mod">
  |<MENOS: "-">
  |<MAS: "+">
  |<MULT: "*">
  |<DIV: "/">
  |<UNION: "~">
  |<TER: ("@" )+

```

```

  |("&. ")+
  |("["")+
  |("\ ")+
  |("?")+
  |("["")+
  |("$")+
  |("#")+
  |("^")+
  |("&")+
  |("-")+
  |("_")+
  |("/")+
  |("[")+
  |("]")+
  |("|")+
  |("!"")+
  |("|")+
  |("&")+
  |("!"")+
  |("\'|"([ "A"- "Z"])|([ "a"- "z"]))) +
  >
}

```

La estructura fue declarada mediante una expresión regular donde identifica las declaraciones o condiciones o ciclos donde pueden ser repetido infinita de veces, así como ninguna.

Principal ()

```
((INICIO | TER | EPSILON (ROMPER ARD0()) | (NUMERO ARD1()) | (CADENA ARD2()) | (BOOLEAN ARD3()) | (IDENT ExistenciasIDsConRedir()) | (SI ARD4()) | (BUCLE ARD5()) | (REPETIR ARD6()) | (ENTR ETA()) | TER | (IMPR IPA()))* (FIN | TER | EPSILON)
```

En esta parte del código se encuentran los métodos que se utilizan para realizar las declaraciones, los cuales se apoyan en la gramática.

Cada tipo de dato que detecta nuestro compilador es evaluado con relativamente la misma gramática, pero variando en el método final en cada tipo de dato.

Cada vez que se lea el token TER se muestre un mensaje de error léxico, y al leer el token EPSILON se muestre un mensaje de error sintáctico.

Token Romper

<ROMPER>

ARD0() ARD0():

FDI

Token Número

<NUMERO()> ARD1()

ARD1 → IDENTIFICADOR ValidacionIDs() DCG | TER DCG () | EPSILON DCG ()

ValidacionIDs() : " validacion de que no existe otra variable con el mismo nombre declarada"

ingresoIDs : " almacenamiento de identificadores y su tipo"

DCG: ASIGNACION DCD() | FDI | TER | EPSILON

DCD: V_NUM DCC () | IDENT DCC()

DCC: FDI | (MAS DCD()) | (MENOS DCD()) | (MULT DCD()) | (DIV DCD()) | (RESIDUO DCD())

Token String

<STRING()> ARD2()

ARD2(): IDENTIFICADOR ValidacionIDs() DCI() | (TER DCG()) | (EPSILON

DCG()) DCI(): ASIGNACION DCE() | FDI | TER | EPSILON

DCE(): IDENTIFICADOR DCF() | V_ALF_NUM

DCF() DCF: UNION DCE() | FDI

Token Boolean

<BOOLEAN> ARD3()

ARD3(): IDENTIFICADOR DCJ() | TER DCG() | EPSILON

DCG() DCJ(): ASIGNACION DCK() | TER | EPSILON

Token IF

<IF> ARD4()

ARD4: CNA() LLAVEI (Id_Para_Estructuras() | Bucle() | Repetir() | Funciones())*

LLAVED CNA(): VF CNZ() | V_NUM CNB() | V_ALF_NUM CND() | IDENT CNF()

CNB(): MAYORQ CNC() | MENORQ CNC() | IGUAL CNC() | DIFERENTE

CNC() CNC(): V_NUM CNZ() | IDENT CNG() | VF CNZ()

CND(): IGUAL CNE() | DIFERENTE

CNE() CNE(): V_ALF_NUM CNZ |

IDENT CNG()

CNG: "Valida el tipo de dato que está ingresando y sobre eso lo redirige"

CNZ: OR CNA() | AND CNA() | EPSILON

Token Ciclo

<WHILE> ARD5()
ARD5(): ((While() Llavei()) | (Repetir() RPA() Llavei())) Declaraciones() |
Condiciones() | Ciclos()* Llaved()
RPA: (IDENTIFICADOR (RPB() | RPD()) | (V_num RPB()) | (V_ALF_NUM RPD()))
RPB: (MAYORQ RPC()) | (MENORQ RPC()) | (IGUAL RPC()) | (DIFERENTE
RPC()) RPC: (V_NUM RPG()) | (IDENT RPG())
RPD: (IGUAL RPE()) | (DIFERENTE
RPE()) RPE: (V_ALF_NUM RPG()) |
(IDENT RPG()) RPG: ((ASOCIATIVO
(IDENT) RPI())
RPI: (MAS RPJ()) | (MENOS RPJ()) | (MULT RPJ()) | (DIV RPJ()) | (RESIDUO RPJ())
RPJ: (IDENT) | (V_NUM)

Token Leer

<LEER> ETA()
ETA() -> ASOCIATIVO ETB() | EPSILON
ETB() ETB() -> IDENT ETC() | EPSILON
ETC() ETC() -> FDI | EPSILON

Token Imprimir

<IMPRIMIR> IPA()
IPA() -> ASOCIATIVO IPB() | EPSILON IPB()
IPB() -> IDENT IPC() | V_ALF_NUM IPC() | V_NUM IPC() | VF IPC() | TER IPC() |
EPSILON IPC()
IPC -> UNION IPB() | FDI

Condicionales

El compilador solo cuenta con una estructura condicional, la estructura condicional "si" permite ejecutar un bloque de código si se cumple una determinada condición.

- “Cláusula Si” “Condición” “Llave izquierda” “Código” “Llave derecha”.

Dentro de "Código" se hace un bloque de instrucciones, que puede incluir otras condiciones, ciclos, así como también los métodos para recibir y mostrar datos en el programa. En cuanto a las condiciones, éstas pueden consistir en una serie de validaciones que se evalúan como verdaderas o falsas, es decir, como una variable booleana. Además, es posible realizar comparaciones como mayor que, menor que, igual que o diferente que, comparando dos valores. Si deseamos combinar varias condiciones en la misma cláusula, podemos utilizar los operadores lógicos "and" y "or". Estos operadores permiten unir las condiciones para que se cumpla al menos una (o) o todas (y), según sea necesario. Un ejemplo sería el siguiente:

```
si (S == S1) {  
    imprimir ('Las dos palabras son iguales');  
}  
  
si (S != S1) {  
    imprimir ('Las dos palabras son diferentes');  
}
```

Ciclos

Se cuenta con 2 estructuras de ciclos, la primera es la menos compleja y consta solo de 3 pasos (sin contar lo que pueda estar dentro del bucle).

“Para” “Llave izquierda” “Código” “Llave derecha”

La estructura en cuestión seguirá ejecutándose en un ciclo hasta que se detecte la palabra "Romper;", momento en el cual finalizará su ejecución. Si la palabra "Romper;" no es detectada, entonces la estructura seguirá ejecutándose repetidamente.

Para la estructura “Mientras” contamos con una combinación de condición más un proceso:

“Mientras” “Parámetro” “Llave izquierda” “Código” “Llave

derecha” El “parámetro” de la estructura se puede dividir en:

“condición” “dos puntos” “proceso”

Método de entrada

Esta instrucción consta de:

“Leer” “Dos puntos” “Identificador” Punto y

coma” Ejemplo: Leer: Z;

Método de salida

La función que tiene este método es mostrar algún resultado en la pantalla y solo basta con escribir la instrucción:

Imprimir: ‘Adjunte_Valor_a_Imprimir’;

El valor puede ser cualquier tipo de dato, también se permite imprimir más de un dato utilizando el signo “~” (virgulilla), quedando de la siguiente manera:

Imprimir: ‘Mensaje_a_Imprimir’ ~ W ~ 16;

Tabla de errores

En este código pueden aparecer errores que se originan por el uso de símbolos que no pertenecen al uso permitido del alfabeto (error léxico) o por el uso de palabras reservadas en una posición no permitida (error sintáctico). En caso de encontrarse algún símbolo que no corresponda a nuestro lenguaje, se producirá un error léxico según lo indicado en la tabla siguiente.

Tabla de errores por asignación de signos

Tipo	Valor
Num F = 0 +	Lo interpreta como un error en el tecleo el operador aritmético puesto que la espera un punto y coma
Num = +	Esta instrucción no tiene sentido lógico por ende sugiere colocar un punto y coma desde el identificador
Num +	No le encuentra lógica por lo cual marca un error y sugiere eliminar la línea.

Tabla de errores por punto y coma

Tipo	Valor
Num A	Es tomado como una declaración por ende se espera una terminación con punto y coma
Num A =	Al solo existir un símbolo delante de nuestro identificador lo interpreta como un símbolo agregado erróneamente.
Num A = 5+ ó A = 5+	Es una asignación de un valor incompleto, para la solución rápida es sustituir el símbolo aritmético por un punto y coma.
Num A = 5	Es una instrucción de instancia-asignación sin embargo arroja un error por falta del punto y coma

Los errores que se producen por falta de punto y coma en el código pueden originarse en las instancias y asignaciones, así como también al utilizar palabras reservadas.

Tabla de errores de identificador y palabras reservadas para tipo de datos

Tipo	Valor
Num d = 7 - 1;	Es generado un error léxico debido a que los identificadores van en mayúsculas.
num D = 9+5;	Es generado un error léxico ya que la gramática del compilador indica que las palabras reservadas son iniciadas con mayúscula

Tabla de errores en la condición Si

Tipo	Valor
si A > B	Al haber indicado los parámetros, se necesitan llaves para delimitar el bloque de código que pertenece a dicha condición.
Si A<B	Es marcado como error léxico por la falta de llaves y por la mala escritura de la palabra reservada Si
si A > {Sentencias}	Los parámetros de la condición están incompletos

Tabla de errores del ciclo mientras

Tipo	Valor
mientras{	Es detectado un error por falta de una llave
Mientras{}	Es un error encontrado por la mala escritura de la palabra reservada

Tabla de errores del ciclo para

Tipo	Valor
para i<5: i = i+1	Después de haber indicado los parámetros se necesita delimitar el bloque de sentencias por lo cual se sugiere ingresar las llaves
Para i<2: i={}	Es detectado un error léxico debido a estar mal escrita la palabra reservada y cuenta con un error en los parámetros
para l: l=l+1{}	Los parámetros del ciclo están incompletos
para l>5: l = l+1{}	Se genera un error léxico al no estar escrita de manera correcta la palabra reservada del ciclo repetir

Tabla de errores para el método leer

Tipo	Valor
leer:	Error ocasionado por la falta de un identificador y también por la falta del punto y coma faltante en la instrucción
Leer a;	Error léxico ocasionado por la mala escritura de la palabra reservada leer, también por la falta de los dos puntos de asignación y por el identificador en minúsculas.
leer;	Error ocasionado por la falta de los dos puntos de asignación y el identificador

Tabla de errores para el método imprimir

Tipo	Valor
imprimir: A	Falta de un punto y coma
imprimir E;	Error encontrado por falta de los dos puntos de asignación
imprimir:'Valor' E;	Error detectado por falta de signo de concatenación para imprimir más de un valor

imprimir: hola';	Error al querer imprimir un texto por falta de un símbolo para la cadena.
------------------	---

Imprimir: 'Hola'	Error léxico en la escritura de la palabra reservada
~ 5;	imprimir;

Sintaxis de sentencia de lenguaje

Sentencia	Descripción	Sintaxis	Ejemplo
Inicio	Sintaxis para iniciar un programa mediante este lenguaje	Inicio código o Fin	Inicio Num B = 15; Fin
Declaración de variables	Sintaxis para declarar las variables	Num m Cad d Bool	Num F; Cad S; Bool Z=Falso;
Asignación de valores	Sintaxis para la asignación de valores en cada tipo de variable	Identificador de la variable = valor de la variable	E = 17;
Imprimir	Sintaxis de impresión de la pantalla	imprimir: ' ' ;	imprimir 'el valores' ~ E;
Condición simple (si)	Sintaxis donde en caso de cumplirse la condición se realizará la instrucción entre las llaves pertenecientes a esa condicional	si Comparación { Instrucciones }	si A >20 { Bloque de instrucciones }
Ciclo para	Sintaxis del ciclo para	para Comparación: asignación {instrucción}	para B > 2: B = B + 1 { Bloque de instrucciones }
Ciclo mientras	Sintaxis del ciclo mientras	mientras{comparación}	mientras{}
Fin	Sintaxis para terminar el programa en este lenguaje	Inicio código o Fin	Inicio Num B = 12; Fin

Análisis semántico

Semántico es la etapa en la que se examina el significado de las estructuras sintácticas y se verifica que sean coherentes y tengan sentido en el contexto del programa. Esta fase se centra en comprender el significado de las instrucciones y expresiones, cómo interactúan con los datos y otros componentes del programa, y si cumplen con las reglas y restricciones del lenguaje de programación.

Tareas principales del análisis semántico:

1. **Verificación de tipos:** Para este caso, el compilador no cuenta precisamente con la validación de tipos, lleva la comprobación de errores semánticos de tipo existente.
2. **Resolución de identificadores:** Durante el análisis semántico, se verifica la existencia y el alcance de las variables y funciones declaradas en el programa. Se debe garantizar que se utilicen identificadores válidos y que no existan conflictos de nombres. Además, se realiza la asociación entre un identificador y su tipo de dato, lo que es esencial para la generación de código.
3. **Coherencia en estructuras de control:** En esta etapa, se verifica que las estructuras de control como condicionales y bucles sean utilizadas de manera adecuada. Por ejemplo, se debe comprobar que las condiciones de los "si" y "mientras" sean expresiones booleanas válidas y que los bucles tengan una condición de finalización clara.
4. **Comprobación de errores semánticos:** El análisis semántico es responsable de detectar errores que no se pueden identificar en las etapas de análisis léxico y sintáctico. Esto incluye errores como asignación de tipos incorrectos, uso de variables no declaradas, o operaciones no permitidas en el contexto del programa.
5. **Generación de información para la generación de código:** Durante el análisis semántico, se recopila la información necesaria para la generación de código intermedio o código objeto. Esto implica almacenar información sobre tipos de datos, identificadores y estructuras de control que serán utilizados en la etapa posterior de generación de código.

Importancia del análisis semántico:

El análisis semántico es esencial para garantizar que un programa sea coherente y correcto en cuanto a su significado y uso de variables. Al realizar una verificación exhaustiva de tipos y resolución de identificadores, se reducen los errores y se mejora la calidad del código generado. Además, permite al compilador realizar optimizaciones más efectivas y producir un código más eficiente.

Para llevar a cabo un análisis completo del código, se deben incluir comprobaciones adicionales que van más allá del simple reconocimiento de una cadena dentro del lenguaje en las fases de análisis léxico y sintáctico. Estas comprobaciones adicionales se realizan en la fase de análisis semántico para garantizar que el programa sea coherente y se pueda compilar correctamente.

Las dos tareas principales del análisis semántico son: la verificación de tipos y la conversión a una representación intermedia.

Durante la Fase Semántica, se incluyeron nuevas variables en el compilador, como un arreglo para almacenar los tipos de errores encontrados, variables para almacenar el par ordenado de Tipo y Valor, y un objeto para guardar de forma secuencial las variables declaradas.

Esto nos ayuda para aquellos errores en donde las variables ya existen. Ejemplo:

```
-----Errores:
Error Semántico. (Lin.4, Col.5). El id ' Y ' ya esta en uso
Compilación Finalizada.
```

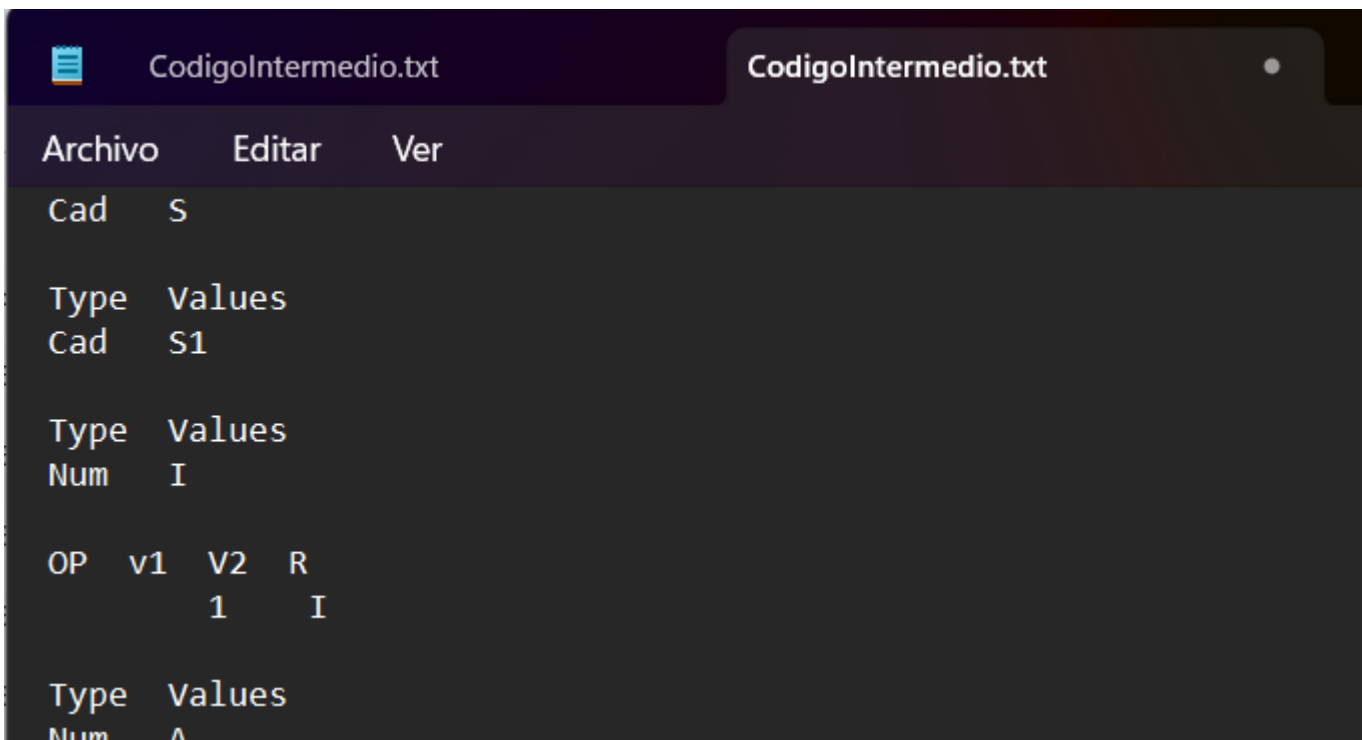
Generación de código intermedio

Se agregó la funcionalidad de la generación un código intermedio que permitirá evitar la necesidad de usar un compilador diferente para cada tipo de máquina.

En este caso de nuestro compilador, se implementó el código de tres direcciones de la notación prefija que nos muestra de lado izquierdo los operadores y de lado derecho los operandos.

Se declararon las siguientes variables; Un String que nos indicará la ruta donde se va a crear el archivo.txt que en el cual se guardará el código intermedio, un objeto de la clase CTDOA y EscrituraCI que permitirán almacenar el código intermedio y guardarlo en el identificador que se estará utilizando.

```
String ruta = "C:/Epsilon/CodigoIntermedio.txt";
CTDOA codigo_aritmetico = new CTDOA();
EscrituraCI ECI = new EscrituraCI(ruta); //Escritura de Código Intermedio
String CI = ""; //Almacén de Código Intermedio
String IGCI = ""; //IDENTIFICADOR en uso para la GENERACIÓN de CÓDIGO INTERMEDIO
int label = 0; //Etiquetas usadas en mientras
boolean aplicaGCI = true; //Valida que la instrucción este correcta para generar el Código Intermedio
boolean vigilanteId = true;
```



```
Archivo  Editar  Ver
Cad      S
Type     Values
Cad      S1
Type     Values
Num      I
OP   v1  V2  R
      1   I
Type     Values
Num      A
```

Manual de usuario

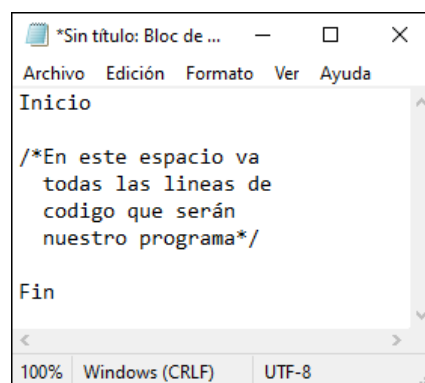
Estructura de la sentencia principal.

Inicio

```
//Código
```

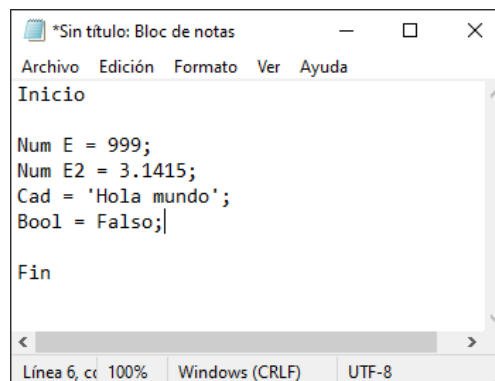
Fin

Como podemos apreciar para poder reconocer un programa, nuestro código de estar ordenado de la manera adecuada dentro de las palabras “inicio” y “fin” que mostraran dónde deben ir las funciones de nuestro programa y en negritas podemos ver cuáles son las partes importantes y fundamentales para poder ejecutar nuestro método principal.



Uso de los tipos de datos dentro del lenguaje.

Hay que reconocer que tipo de dato se usará dentro del lenguaje, es crucial para saber determinar qué es lo que tu programa quiere hacer, como ya se estudió dentro de nuestras palabras reservadas existen algunas que les da sentido a nuestras variables. Esto quiere decir que sin ellas no podríamos reconocer para que estamos creando nuestras variables. Para declarar variables numéricas se utiliza el tipo de dato Num, el cual nos permite crear variables de valor entero y decimal, con el tipo de dato Cad, creamos variables que acepta cadena de caracteres, con el tipo de dato Bool, creamos variables verdadero o falso.



Estructura de declaración de variables.

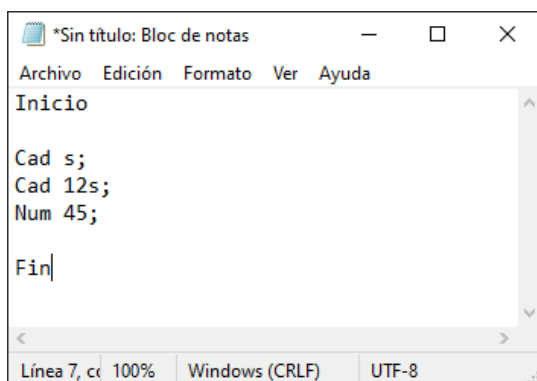
En la declaración de variables se debe hacer referencia al tipo de dato que se desea utilizar, entre ellos se puede poner (NUM, STRING, BOOLEAN). Para que se puedan crear variables se necesitara crear una serie de caracteres los cuales están limitados a utilizar letras de la A – Z solo en mayúsculas. En la declaración de variables solo se podrá utilizar una letra o bien una letra y cualquier número o serie de números después de la misma, esto con el fin de diferenciar si se desea utilizar la misma letra más de una vez.

Ejemplo:

Cad S;	Num E;
Cad S1;	Num E12;

Como vemos esta segunda forma nos permite cambiar el valor de nuestra variable y es importante ya que estas son parte de la mayoría de las sentencias dentro del lenguaje.

Ejemplo programa con variable incorrecta:



```
*Sin título: Bloc de notas
Archivo Edición Formato Ver Ayuda
Inicio

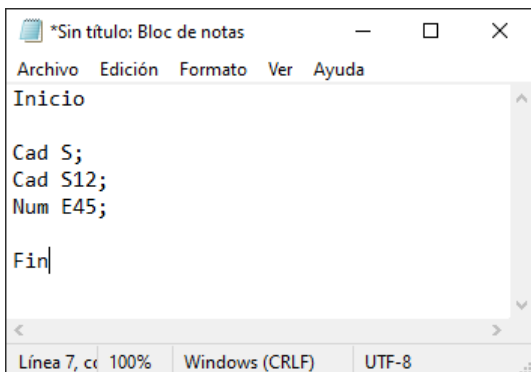
Cad s;
Cad 12s;
Num 45;

Fin|

Línea 7, c1 100% Windows (CRLF) UTF-8
```



Ejemplo programa con variable correcta:



```
*Sin título: Bloc de notas
Archivo Edición Formato Ver Ayuda
Inicio

Cad S;
Cad S12;
Num E45;

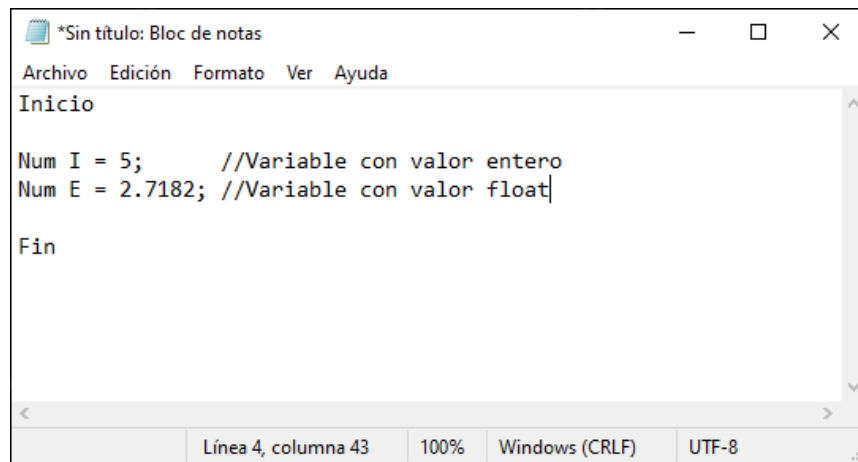
Fin|

Línea 7, c1 100% Windows (CRLF) UTF-8
```



Ejemplo de programa con variables de valores numéricos:

(El tipo de variable Num acepta tanto valores enteros como de punto flotante)



A screenshot of a Notepad window titled "*Sin título: Bloc de notas". The window has a menu bar with "Archivo", "Edición", "Formato", "Ver", and "Ayuda". The text content is as follows:

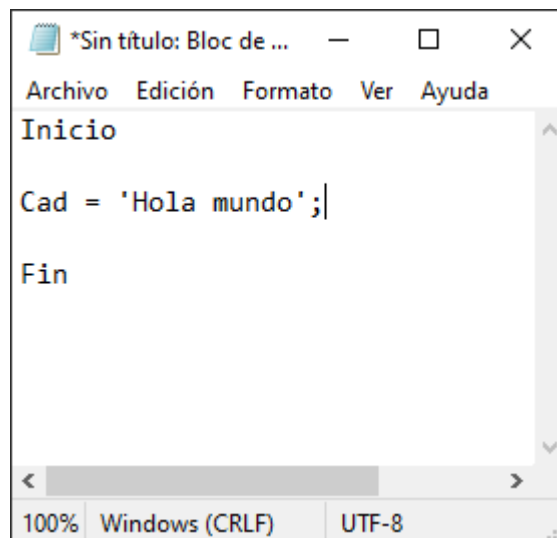
```
Inicio

Num I = 5;      //Variable con valor entero
Num E = 2.7182; //Variable con valor float

Fin
```

The status bar at the bottom shows "Línea 4, columna 43", "100%", "Windows (CRLF)", and "UTF-8".

Ejemplo de programa con variables de valores de caracteres:



A screenshot of a Notepad window titled "*Sin título: Bloc de ...". The window has a menu bar with "Archivo", "Edición", "Formato", "Ver", and "Ayuda". The text content is as follows:

```
Inicio

Cad = 'Hola mundo';

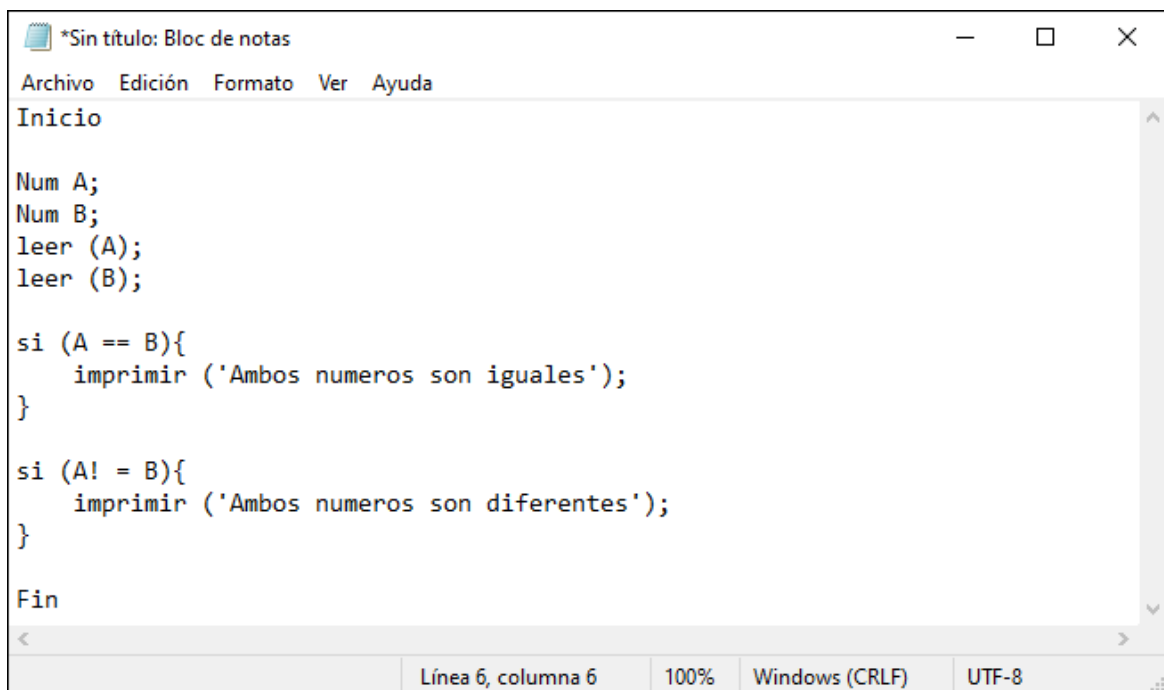
Fin
```

The status bar at the bottom shows "100%", "Windows (CRLF)", and "UTF-8".

Estructura de la sentencia “SI”.

```
si ( //Sentencia lógica) {  
  
    //Funciones  
  
}
```

Como observamos hasta esta parte de la sentencia ‘si’ podemos percatarnos que esta cuenta con palabras reservadas y símbolos cuyo orden es fundamental para poder ejecutarse de manera adecuada ya que si no genera un error en el orden de la estructura principal del código. En cuanto al apartado de las funciones estas se deben checar que las variables tengan cierta coherencia a la hora de ser ejecutado, ya que sin ella existiría cierto tipo de error el cuál no nos dejara concluir la sentencia.



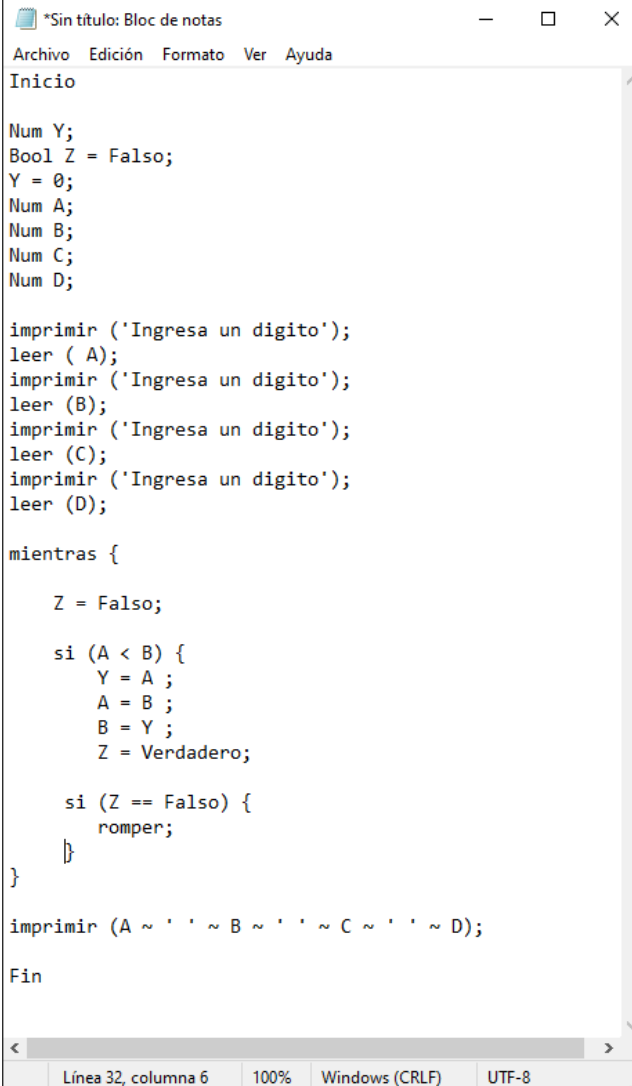
```
*Sin título: Bloc de notas  
Archivo Edición Formato Ver Ayuda  
Inicio  
  
Num A;  
Num B;  
leer (A);  
leer (B);  
  
si (A == B){  
    imprimir ('Ambos numeros son iguales');  
}  
  
si (A! = B){  
    imprimir ('Ambos numeros son diferentes');  
}  
  
Fin  
  
Línea 6, columna 6 100% Windows (CRLF) UTF-8
```

Estructura mientras.

Esta cuenta con la parte de generar bucles dentro de nuestro lenguaje esta se apoya de las variables y demás funciones. Esta estructura es un bucle infinito hasta que se encuentre con un BREAK o “romper”.

```
mientras {  
  
    //Funcio  
    nes  
    romper;  
  
}
```

Como podemos apreciar esta se forma de manera más sencilla ya que trabaja directamente con los componentes de esta misma y no sufre cambio alguno para poder ser utilizada.



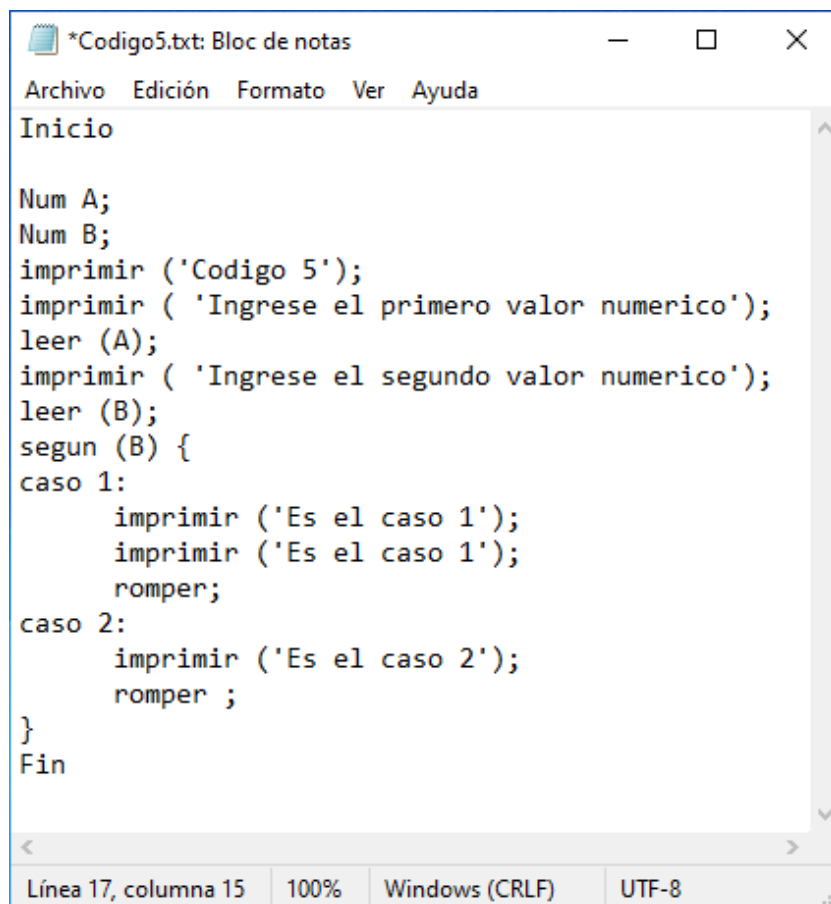
```
*Sin título: Bloc de notas  
Archivo Edición Formato Ver Ayuda  
Inicio  
  
Num Y;  
Bool Z = Falso;  
Y = 0;  
Num A;  
Num B;  
Num C;  
Num D;  
  
imprimir ('Ingresa un digito');  
leer ( A);  
imprimir ('Ingresa un digito');  
leer (B);  
imprimir ('Ingresa un digito');  
leer (C);  
imprimir ('Ingresa un digito');  
leer (D);  
  
mientras {  
  
    Z = Falso;  
  
    si (A < B) {  
        Y = A ;  
        A = B ;  
        B = Y ;  
        Z = Verdadero;  
  
        si (Z == Falso) {  
            romper;  
        }  
    }  
}  
  
imprimir (A ~ ' ' ~ B ~ ' ' ~ C ~ ' ' ~ D);  
  
Fin
```

Línea 32, columna 6 100% Windows (CRLF) UTF-8

Estructura según – caso

Es un tipo de mecanismo de control de selección utilizado para permitir que el valor de una variable o expresión cambie el flujo de control de la ejecución del programa mediante búsqueda y mapa.

```
segun (//Variable) {  
    caso 1:  
        //Funciones  
  
    caso 2:  
        //Funcio  
nes romper;  
}
```



The screenshot shows a Notepad window titled '*Codigo5.txt: Bloc de notas'. The menu bar includes 'Archivo', 'Edición', 'Formato', 'Ver', and 'Ayuda'. The text content is as follows:

```
Inicio  
  
Num A;  
Num B;  
imprimir ('Codigo 5');  
imprimir ( 'Ingrese el primero valor numerico');  
leer (A);  
imprimir ( 'Ingrese el segundo valor numerico');  
leer (B);  
segun (B) {  
    caso 1:  
        imprimir ('Es el caso 1');  
        imprimir ('Es el caso 1');  
        romper;  
    caso 2:  
        imprimir ('Es el caso 2');  
        romper ;  
}  
Fin
```

The status bar at the bottom indicates 'Línea 17, columna 15', '100%', 'Windows (CRLF)', and 'UTF-8'.

Estructura for.

La estructura “for” se usa en aquellas situaciones en las cuales conocemos la cantidad de veces que queremos que se ejecute el bloque de instrucciones.

La ejecución de esta estructura de control es la siguiente:

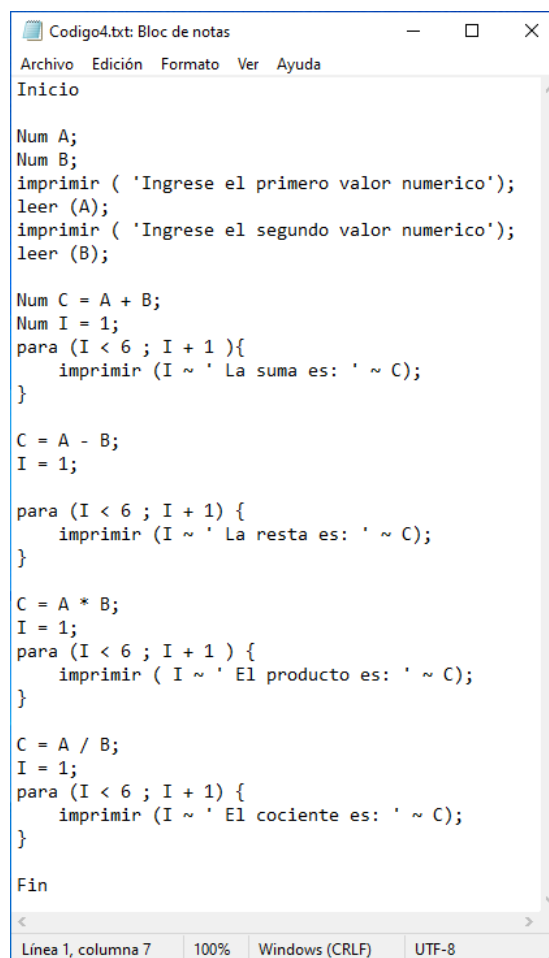
Se establece el valor inicial de la variable de control definida en la asignación inicial. Evalúa la condición de continuación:

- Si el resultado es true se ejecuta el bloque de sentencias, se efectúa el cambio de la variable de control y se evalúa nuevamente la condición de continuación;
- Si el resultado es false el bucle se termina.

Para (*/*sentencia lógica*/* ; */*identificador*/* + 1) {

//Funciones

}



```
Codigo4.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Inicio

Num A;
Num B;
imprimir ( 'Ingrese el primero valor numerico');
leer (A);
imprimir ( 'Ingrese el segundo valor numerico');
leer (B);

Num C = A + B;
Num I = 1;
para (I < 6 ; I + 1 ){
    imprimir (I ~ ' La suma es: ' ~ C);
}

C = A - B;
I = 1;

para (I < 6 ; I + 1) {
    imprimir (I ~ ' La resta es: ' ~ C);
}

C = A * B;
I = 1;
para (I < 6 ; I + 1 ) {
    imprimir ( I ~ ' El producto es: ' ~ C);
}

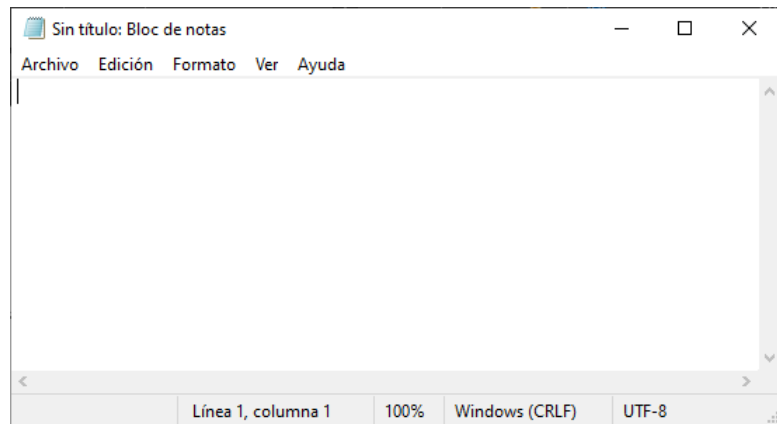
C = A / B;
I = 1;
para (I < 6 ; I + 1) {
    imprimir (I ~ ' El cociente es: ' ~ C);
}

Fin

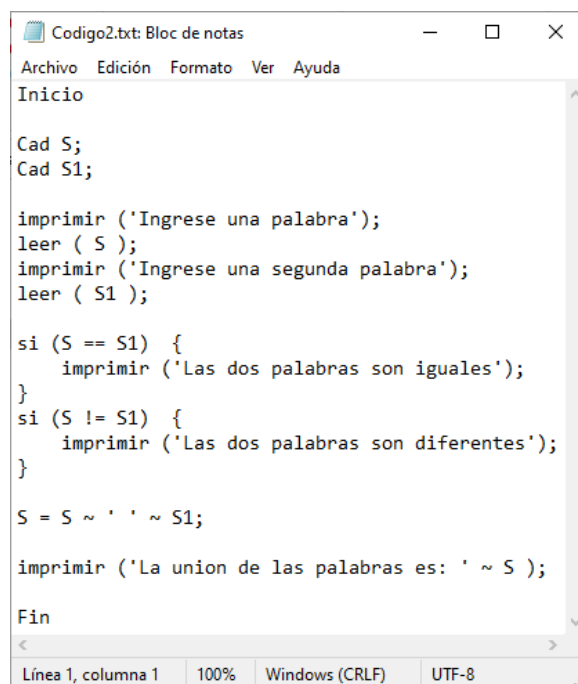
Línea 1, columna 7 100% Windows (CRLF) UTF-8
```

Ejecución de programa.

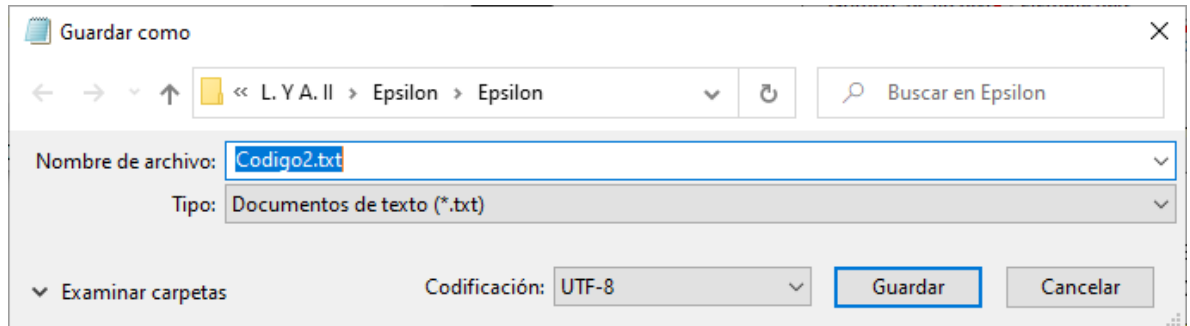
1. Abrir un block de notas, es aquí donde vamos a escribir el código del programa que deseemos hacer.



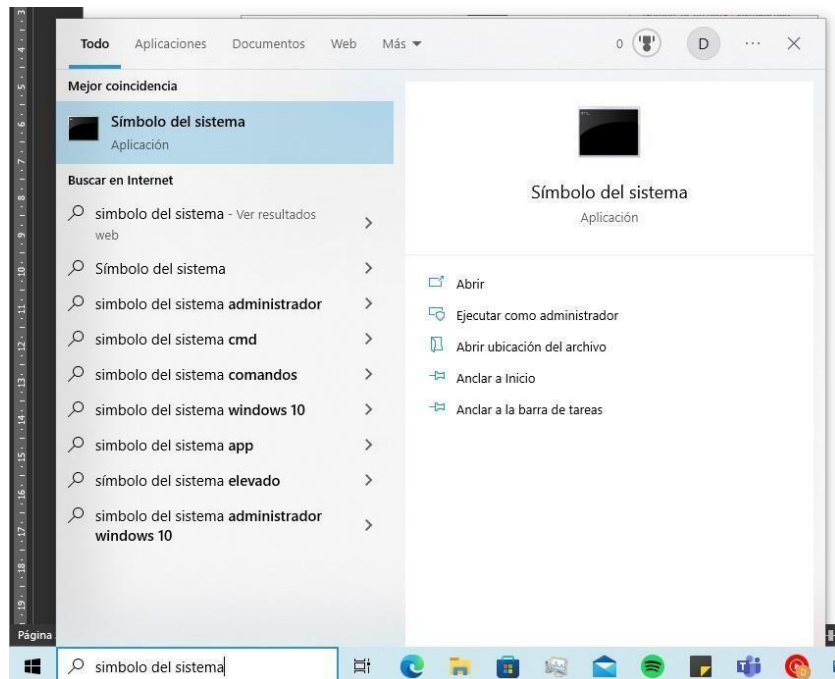
2. Considérese que al momento de hacer la declaración de sus variables se haga de la manera correcta según las especificaciones del lenguaje, de lo contrario al momento de la ejecución se mostrará un error en la misma.
3. Dentro del block de notas se comenzará con la realización de nuestro programa escribiendo el código de lo que se desee programar. Hay que recordar que todas las instrucciones que se programen deben estar dentro de las sentencias "Inicio" y "Fin".



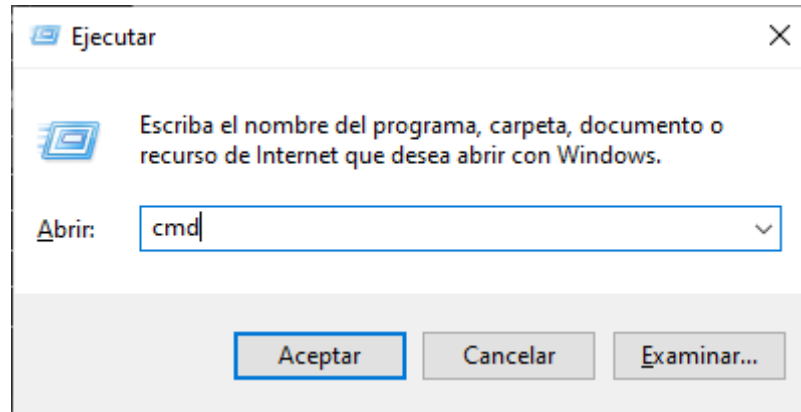
4. Terminado ya nuestro código del programa que hayamos hecho se debe guardar el archivo con el nombre que lo vayamos a identificar. Este debe ser guardado con la extensión “.txt”, que es la que usaremos a la hora de la compilación.
- Recordar muy bien en donde se guarda este programa para hacer más fácil el proceso de compilación sin tener la necesidad de acudir a buscar donde se guardó.



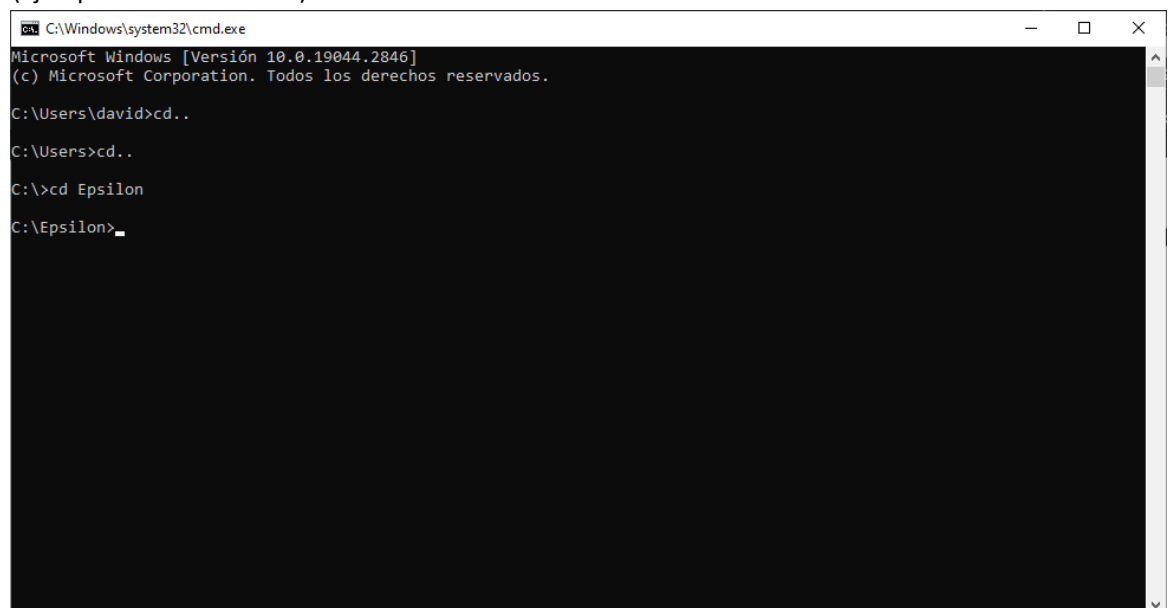
5. Teniendo listo todo lo anterior, estamos listos para compilar. No dirigimos al menú inicio de nuestro sistema operativo y buscamos la consola o símbolo del sistema.



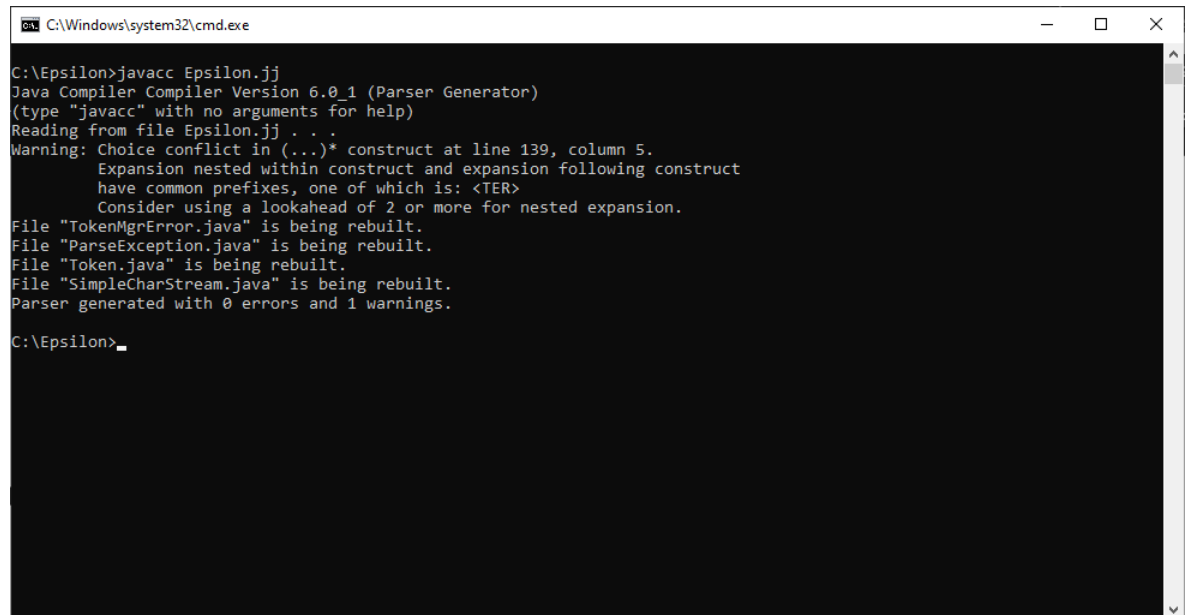
También podemos acceder de manera más eficiente haciendo la combinación de teclas **Win + R**, lo que nos hará aparecer una ventana emergente donde escribiremos el comando "cmd" el cual nos abrirá la consola.



6. Una vez dentro de nuestro símbolo del sistema nos tendremos que dirigir a la carpeta donde tenemos nuestro compilador que es donde están los archivos necesarios para la ejecución. Los comandos necesarios para acceder dependerán de la ruta donde tengamos ubicado nuestro compilador.
(Ejemplo de nuestro caso).



7. Una vez dentro de nuestra carpeta donde se ubica el compilador, es necesario ejecutar el comando “javacc” seguido del nombre del archivo con extensión “.jj”.
(Ejemplo de nuestro caso).

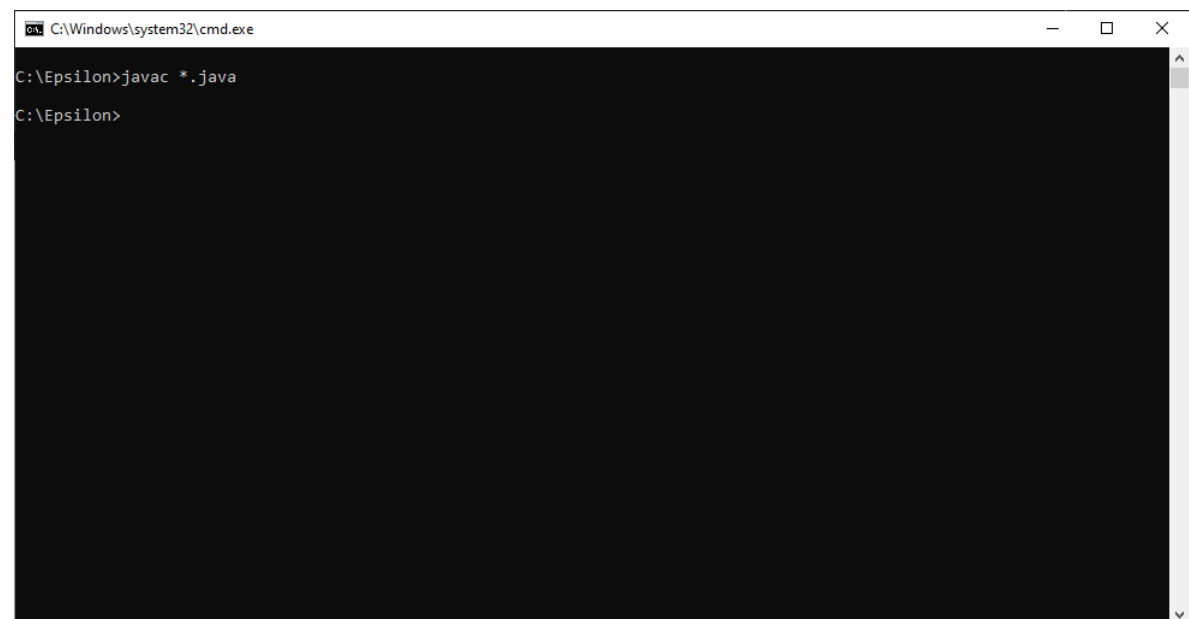


```
C:\Windows\system32\cmd.exe

C:\Epsilon>javacc Epsilon.jj
Java Compiler Compiler Version 6.0_1 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Epsilon.jj . . .
Warning: Choice conflict in (...) * construct at line 139, column 5.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: <TER>
         Consider using a lookahead of 2 or more for nested expansion.
File "TokenMgrError.java" is being rebuilt.
File "ParseException.java" is being rebuilt.
File "Token.java" is being rebuilt.
File "SimpleCharStream.java" is being rebuilt.
Parser generated with 0 errors and 1 warnings.

C:\Epsilon>_
```

8. Una vez ejecutado de manera correcta, lo siguiente es realizar la ejecución de los archivos con extensión “.class”, para lo que utilizaremos el comando “javac *.java”, el cual ejecutara todos los archivos con dicha extensión.

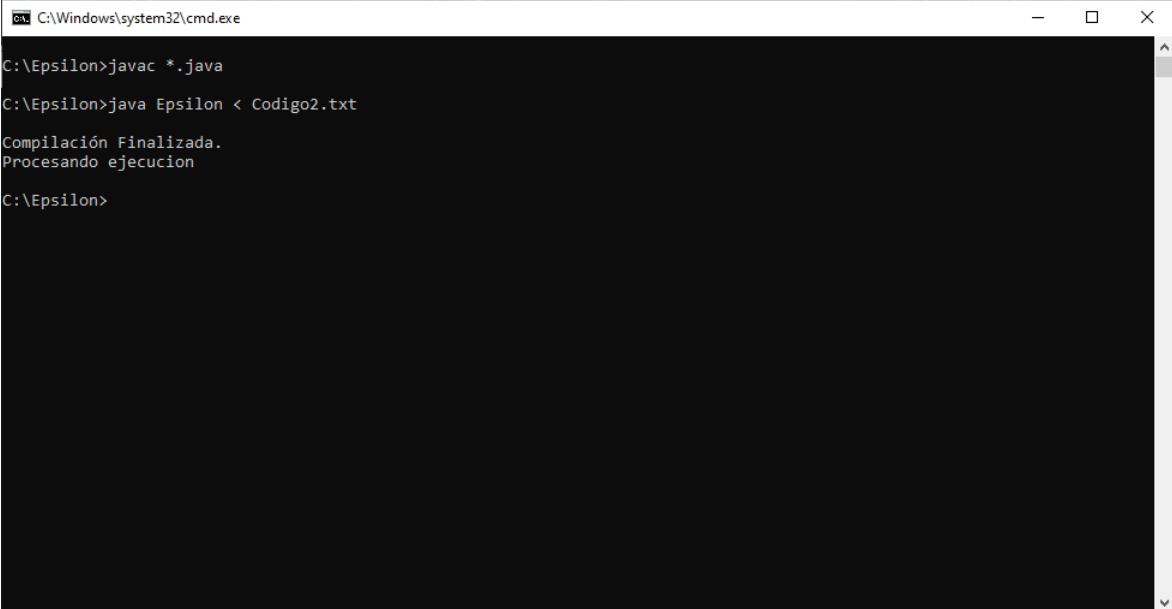


```
C:\Windows\system32\cmd.exe

C:\Epsilon>javac *.java
C:\Epsilon>
```

No nos debe saltar algún error o advertencia, de lo contrario se tiene que hacer una revisión.

9. Con todo lo anterior correctamente realizado, ya se está listo para ejecutar el comando que nos va a compilar nuestro programa. Este comando se compone por la siguiente estructura:
"java NombreArchivo < NombreEjemplo.txt"



```
C:\Windows\system32\cmd.exe
C:\Epsilon>javac *.java
C:\Epsilon>java Epsilon <Codigo2.txt
Compilación Finalizada.
Procesando ejecucion
C:\Epsilon>
```

Conclusión.

La creación de un compilador suele ser de vital importancia para un programador que desea generar su propio lenguaje el cual le puede ser muy valioso en dado caso de que algún lenguaje se le pueda llegar complicar en algún momento y así podrá entender de una mejor manera la creación de un lenguaje. Así como también el mismo podrá declarar de manera única el uso de su lenguaje en cuanto a variables, sentencias entre otras cosas.