

TECNOLÓGICO NACIONAL DE MÉXICO

**INSTITUTO**

**TECNOLÓGICO NACIONAL DE VERACRUZ**

**Compilador “Master” en fases léxico y sintáctico**

Materia: Lenguajes y autómatas I

**Docente: Ing. Hernández Silva José**

Integrantes:

- Gerardo Galván Chávez ---------------E17021510

-Rafael Antonio Gonzalez Zamora-----E17021892

-Jorge Antonio Pedroza Rendon----E17021588

-Pablo Saúl Campos Ricalde---------E17021602

**Introducción:**

En el siguiente documento se describirá el uso del compilador **“Master”,** cual es un compilador desarrollado en JavaCC del lenguaje básico de programación “**Cobra”**.

En este documento se engloban aspectos del manual del sistema, así como también del manual de usuario. Cabe decir que el compilador que se desarrolló, cuenta únicamente con las dos primeras fases de un compilador, la fase léxica y la fase sintáctica. Se detallan en primer lugar los aspectos generales del lenguaje **“Cobra**” como; palabras reservadas, tipos de datos, estructuras condicionales y ciclos, dicho lo anterior, para una compilación correcta en el compilador “**Master”** es necesario conocer la sintaxis del lenguaje **“Cobra”.**

Este documente se describe detalladamente el uso y las estructuras que definen la sintaxis del lenguaje como, las expresiones regulares y las gramáticas libres de contexto.

Definición del proyecto

Como parte de la materia de lenguajes y autómatas, se presenta el siguiente proyecto de la creación de un compilador en las primeras dos fases.

* **Nombre del compilador:** “Master”

Para este compilador fue necesario la creación de un lenguaje de programación básico basado en el lenguaje Python, con el fin de probar las primeras dos fases del compilador.

* **Nombre del lenguaje: “**Cobra**”**

**Descripción del lenguaje “cobra”**

Utilización del idioma inglés, en combinación con algunos símbolos que nos permiten crear los comandos necesarios para que el programa pueda reconocerlos y ejecutar las líneas de código para su fácil aprendizaje y rápida ejecución.

**Diferencia entre *Master y* *Cobra***

*Master*, es el nombre del compilador, el nombre del proyecto que se desarrolló para la materia, en cambio *Cobra* es el lenguaje que se implementa en el compilador; por lo tanto, el compilador *Master* implementa el lenguaje *Cobra* para poder crear su estructura.

Léxico del lenguaje

Para poder utilizar correctamente el compilador será necesario entender correctamente el uso del lenguaje, así como sus estructuras, con esto se ocuparán las gramáticas necesarias para darle sentido al lenguaje.

* **Definición del lenguaje Cobra**

Sentencias de inicio y fin.

|  |  |
| --- | --- |
| Estructura | Valor |
| <DEF><IDENTIFIER><COLON> | Inicio |
| <END> | fin |

Tipos de datos.

|  |  |
| --- | --- |
| Estructura | Valor |
| <CADENA\_STRING> | String |
| <NUMBER> | Numérico |

Tipos de datos booleanos.

|  |  |
| --- | --- |
| Estructura | Valor |
| <TRUE> | Verdadero |
| <FALSE> | Falso |

Concatenación.

|  |  |
| --- | --- |
| Estructura | Valor |
| <CONCATENACION> | & |

Estructuras condicionales

|  |  |
| --- | --- |
| Estructura | Valor |
| <IF> | Condición if |
| <ELSE> | Condición else |
| <ELIF> | Condición else if |
| <SWITCH> | Condición múltiple switch |
| <CASE> | Casos de switch |
| <DEFAULT\_CASE | Caso por defecto |
| <BREAK> | Sentencia break |
| <RETURN> | Sentencia return |
| <IN> | Sentencia for in |

Ciclos

|  |  |
| --- | --- |
| Estructura | Valor |
| <WHILE> | Ciclo while |
| <FOR> | Ciclo for |

Entrada y salida

|  |  |
| --- | --- |
| Estructura | Valor |
| <PRINT> | Salida de datos |
| <INPUT> | Entrada de datos |
| <EOL> | Salto de linea |

Símbolos

|  |  |
| --- | --- |
| Estructura | Valor |
| <LPAREN> | ( |
| <RPAREN> | ) |
| <LBRACKET> | [ |
| <RBRACKE> | ] |
| <COLON> | : |
| <LBRACE> | { |
| <RBRACE> | } |
| <COMMA> | , |
| <DOUBLECOMMA> | “ ” |

Operadores de comparación

|  |  |
| --- | --- |
| Estructura | Valor |
| <EQ> | == |
| <BIG> | > |
| <LESS> | < |
| <LESSEQ> | <= |
| <BIGEQ> | => |
| <NOTEQ> | != |

Operadores lógicos

|  |  |
| --- | --- |
| Estructura | Valor |
| <AND> | and |
| <OR> | or |
| <NOT> | not |

Operadores aritméticos

|  |  |
| --- | --- |
| Estructura | Valor |
| <PLUS> | + |
| <MINUS> | - |
| <TIMES> | \* |
| <DIVIDE> | / |
| <INCR> | ++ |
| <DECR> | -- |
| <ASIGNACION> | = |
| <PLUS\_EQ> | += |
| <MINUS\_EQ> | -= |
| <TIMES\_EQ> | \*= |
| <DIVIDE\_EQ> | /= |

**Identificadores**

Para crear un identificador se hace uso de la expresión regular:

<IDENTIFIER> ::= ["a"-"z","A"-"Z"](["a"-"z","A"-"Z","0"-"9","\_"])

|  |  |
| --- | --- |
| Estructura | Valor |
| <IDENTIFIER> | Ejemplo: Ped123fre |

**Números**

Para el uso de números en identificadores que manejen datos numéricos pueden ser declarados de la siguiente manera:

<NUMBER> ::= (["0"-"9"]+)[(["0"-"9"]+)"."(["0"-"9"]+)][(["0"-"9"]+)"."]["."(["0"-"9"]+)]

|  |  |
| --- | --- |
| Estructura | Valor |
| <NUMBER> | Ej. 100, 0.1, 4., .9 |

## Token <EOL> y <EOLW>

Cobra es un lenguaje que se basa en la escritura limpia de código, el carácter <EOL> indica cuando se ha declarado una instrucción, la declaración de una sentencia y la declaración de funciones y por lo tanto es necesario insertar un salto de línea “\n” en sistemas basados en unix, en sistemas de Windows el formato de salto de línea está dado por “\r\n”.

**Token <EOF>**

Existe un token especial que nos brinda JavaCC, este es el token de tipo <EOF> , el cuál cada vez que se lee, indica que ya no hay más valores dentro del código, por lo que lee espacios en blanco o vacíos, debido a que se consumieron todos los valores dentro del código pasado por el archivo de texto.

Sintaxis del lenguaje

# Estructura de un programa

La estructura básica de un programa está definida por:

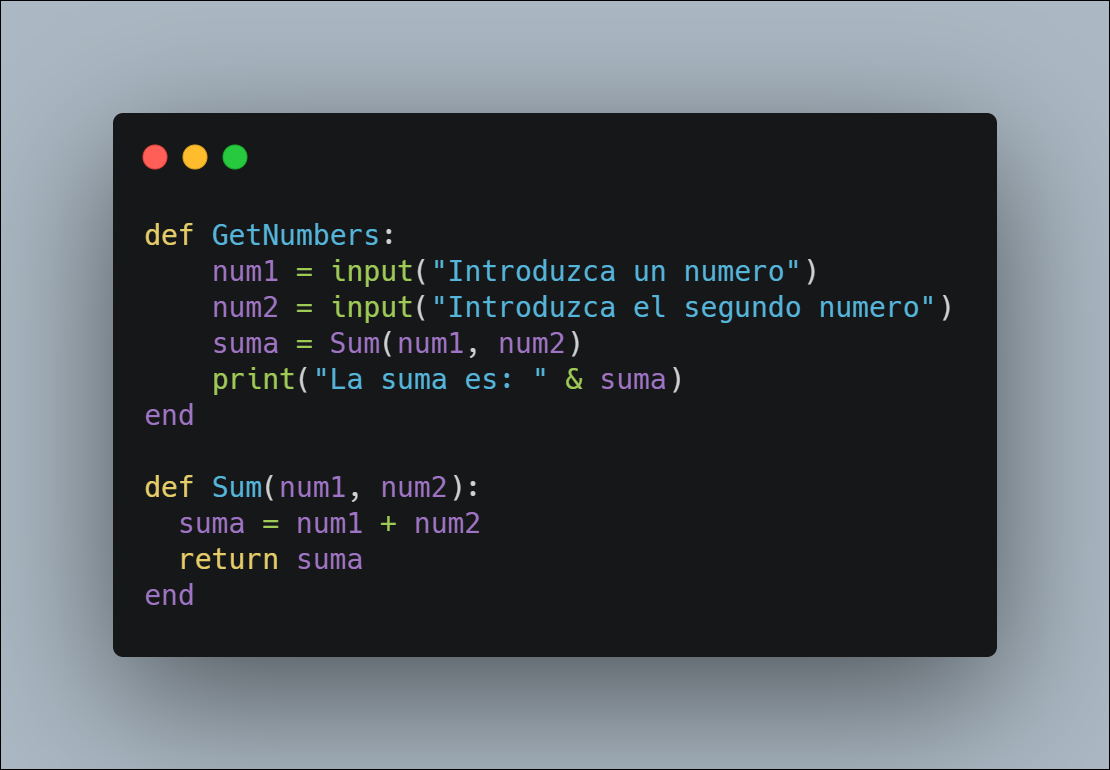
<Programa> ::= (<DEF><IDENTIFIER>[<LPAREN>Args()<RPAREN>]<COLON><EOL>

Code( )

<END>[(<EOL>|<EOLW>)+]

)+

<EOF>



# Argumentos

Cobra permite enviarle argumentos opcionales a una función para que la función los pueda usar.

Args ::= <IDENTIFIER>[(<COMMA><IDENTIFIER>)+]<RPAREN>



# Uso de variables

En cobra las variables no requieren de una definición del tipo de dato, las variables pueden guardar números, cadenas, valores booleanos

Declaración de una variable y asignación:

<IDENTIFIER> ::= <ASIGNACION>(<NUMBER>|<IDENTIFIER>)<EOL>

Declaración de una variable y operación:

<IDENTIFIER> ::= <ASIGNACION>opreracion()

Declaración de variable de texto

<IDENTIFIER> ::= <ASIGNACION><DOUBLECOMMA>(<IDENTIFIER>|<NUMBER>)\*<DOUBLECOMMA><EOL>



Declaración de array:

(<IDENTIFIER> :: = <ASIGNACION><LBRACKET>((<NUMBER>|<IDENTIFIER>)|((<COMMA><NUMBER>)|(<COMMA><IDENTIFIER>)))\*<RBRACKET><EOL>)| (<IDENTIFIER><LBRACKET><NUMBER><RBRACKET><ASIGNACION>(<NUMBER>|<IDENTIFIER>)<EOL>)| (<IDENTIFIER><LBRACKET><IDENTIFIER><RBRACKET><EOL>)



# Sentencias

## Sentencia if

<IF> ::= <IF>Comparacion()<COLON>

Code()

[Else()|Elif()]

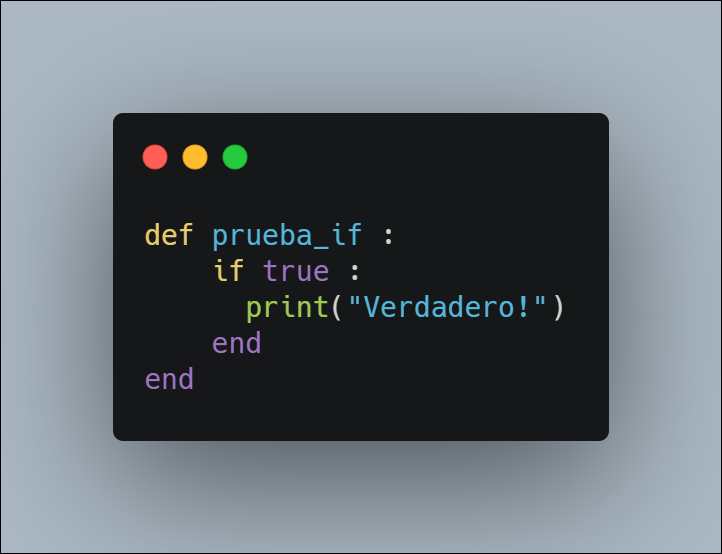
<END>

### Comparación

Una comparación puede ser usada en una sentencia if y puede haber 3 tipos de comparaciones:

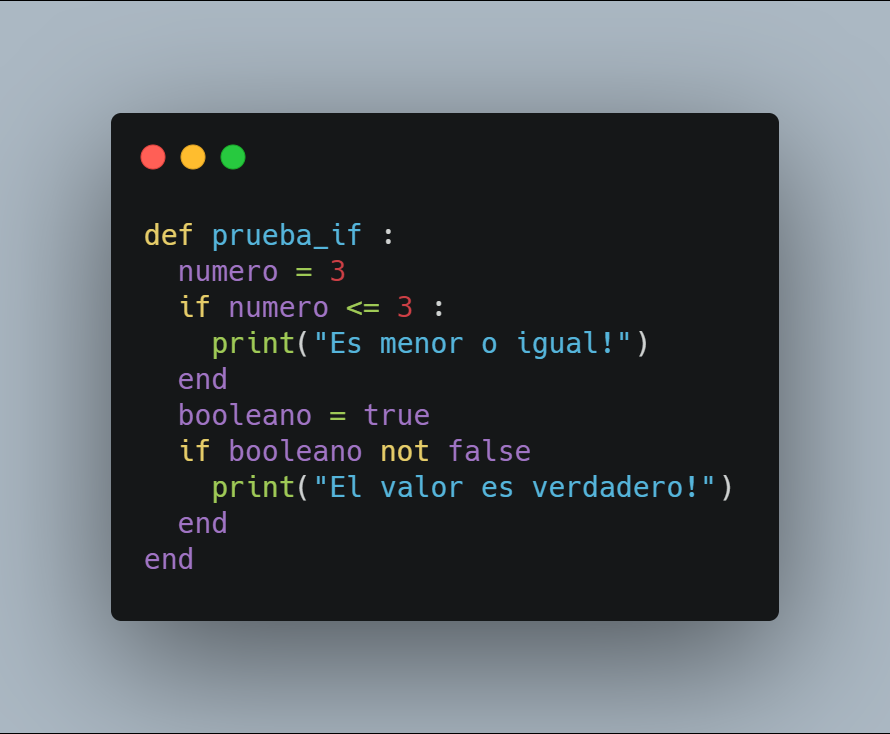
### Comparación simple

<Comparacion\_simple> ::= (<TRUE>|<FALSE>)



### Comparación booleana

<Comparacion\_booleana> ::= ((<NUMBER>)|<TRUE>|<FALSE>|<IDENTIFIER>)(<LESS>|<LESSEQ>|<BIG>|<BIGEQ>|<NOT>|<EQ>)(<TRUE>|<FALSE>|<IDENTIFIER>|<NUMBER>)



### Comparación booleana extendida

<Comparación\_booleana\_extendida> ::= (((<NUMBER>)+|<TRUE>|<FALSE>|<IDENTIFIER>)(<LESS>|<LESSEQ>|<BIG>|<BIGEQ>|<NOT>)((<NUMBER>)+|<TRUE>|<FALSE>|<IDENTIFIER>)[(<OR>|<AND>)])+

**Ejemplo**

If a < b or b > c

## Sentencia else

La sentencía else puede ser usada despues de un if indicando que si una condición no se cumple se hará otra acción

<ELSE> ::= <COLON><EOL>Code()

## Sentencia elif

La sentencia elif puede ser usada después de un if indicando que si una condición evaluada no se cumple entonces se volverá a hacer otra evaluación y en caso de cumplirse se deberá realizar una acción, esta sentencia puede ser repetida distintas veces dentro de una sentencia if.

<ELIF> ::= (Comparacion()<EOL>Code())+

**Sentencia switch**

La sentencia switch es usada para múltiples vías de instrucciones simples, con ella se pretende poder contemplar diferentes casos.

<SWITCH> ::=

<SWITCH><LPAREN><IDENTIFIER><RPAREN><COLON>

(<EOL>|<EOLW>) (<CASE>(<IDENTIFIER>|<NUMBER>)<COLON>(<EOL>|<EOLW>)

Code()<BREAK>(<EOL>|<EOLW>))+ ((<DEFAULT\_CASE><COLON>(<EOL>|<EOLW>)Code()<BREAK>(<EOL>|<EOLW>)<END>(<EOL>|<EOLW>))|<END>(<EOL>|<EOLW>))

### Sentencia print

<PRINT> ::= <PRINT><LPAREN>Imprimible()[{&Imprimible()}]<RPAREN>

### Imprimible

Un imprimible es un un token que puede tomar la forma de una cadena de texto o del valor almacenado en cualquier identificador, dado de la siguiente manera:

<IMPRIMIBLE> :: = <IDENTIFIER>|<DOUBLECOMMA><CHAR>\*<DOUBLECOMMA>