



**Universidad Nacional de Ingeniería**

FACULTAD DE CIENCIAS

ESCUELA PROFESIONAL DE CIENCIAS DE LA COMPUTACIÓN

# SPANISH PSEUDOCODE LANGUAGE

Autores:

Penadillo Lazares Wenses Johan

Bazan Turin Kenjhy Javier

Profesor:

Jaime Osorio Ubaldo

Abril 2023

# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Problemática . . . . .	1
1.2. Objetivos . . . . .	1
1.2.1. Objetivo general . . . . .	1
1.2.2. Objetivos específicos . . . . .	1
1.3. Alcance y limitaciones . . . . .	2
1.3.1. Resultados esperados . . . . .	2
1.3.2. Delimitación del proyecto . . . . .	2
1.4. Justificación y viabilidad . . . . .	2
<b>2. Marco teórico</b>	<b>3</b>
2.1. Lenguajes de programación . . . . .	3
2.2. Sintaxis de los lenguajes de programación . . . . .	3
2.3. Pseudocódigo . . . . .	3
2.4. Enseñanza de la programación . . . . .	3
2.5. Compiladores . . . . .	4
2.6. Entornos de desarrollo integrados (IDE) . . . . .	4
2.7. Código Intermedio . . . . .	4
2.8. Código abierto . . . . .	4
2.9. LLVM (Low-Level Virtual Machine) . . . . .	4
2.10. Método Shift-Reduce . . . . .	4
2.11. Tabla de Símbolos . . . . .	5
2.12. Tabla de Códigos . . . . .	5
<b>3. Estado del arte</b>	<b>6</b>
3.1. DEFINICIÓN DEL LENGUAJE DE PROGRAMACIÓN EPLOAM PARA LA EJE- CUCIÓN DE PSEUDOCÓDIGO Y SU COMPILADOR . . . . .	6
3.2. Compilador y traductor de pseudocódigo para la lógica de programación (CompiPro- gramación) . . . . .	6

3.3. COMPILADOR DE PSEUDOCÓDIGO COMO HERRAMIENTA PARA EL APREN- DIZAJE EN LA CONSTRUCCIÓN DE ALGORITMOS . . . . .	6
3.4. HITO: Pseudocode compiler with graphics library . . . . .	7
<b>4. Metodología</b>	<b>8</b>
4.1. Herramientas . . . . .	8
4.2. Diseño del lenguaje . . . . .	8
4.2.1. Patrón de los tokens y palabras reservadas . . . . .	8
4.2.2. Definición de la gramática . . . . .	11
4.3. Implementación del lenguaje . . . . .	14
4.3.1. Implementación de los tokens con Lex y Python . . . . .	14
4.3.2. Implementación de la gramática con Yacc y Python . . . . .	14
4.3.3. Implementación de nodos básicos del AST . . . . .	14
4.3.4. Implementación del generador de código intermedio . . . . .	14
4.4. Diseño del IDE . . . . .	15
4.5. Implementación del IDE . . . . .	15
4.6. Nombre del compilador . . . . .	15
<b>5. Experimentos y resultados</b>	<b>16</b>
5.1. Procedimiento . . . . .	16
5.2. Pruebas . . . . .	16
5.2.1. Árboles de sintaxis abstracta . . . . .	19
5.2.2. Método Shift/Reduce para verificar la validez de los caracteres de entrada . . .	22
5.2.3. Tabla de Símbolos . . . . .	24
5.2.4. Tabla de Códigos . . . . .	25
5.3. Resultados . . . . .	26
5.4. Análisis . . . . .	26
5.4.1. Características . . . . .	26
5.4.2. Ventajas . . . . .	26
<b>Referencias</b>	<b>28</b>

# Capítulo 1

## Introducción

### 1.1. Problemática

Existe un gran porcentaje de alumnos los cuales no cuentan con un dominio del idioma inglés, muchos de ellos no tienen ni nociones básicas de este idioma. Esto sumado a la dificultad que significa comprender la sintaxis de algunos lenguajes de programación, complica la enseñanza y el aprendizaje de la programación de algoritmos y programación en general. Es debido a esto que se plantea la idea de crear un nuevo lenguaje de programación en español con sintaxis similar a pseudocódigo, buscando así que se facilite esta tarea tanto para alumnos como para profesores.

### 1.2. Objetivos

#### 1.2.1. Objetivo general

Crear un nuevo lenguaje de programación en español con una sintaxis similar a un pseudocódigo, así mismo el compilador y un IDE perteneciente a este lenguaje. De modo que se facilite la enseñanza de la programación de algoritmos para alumnos que no comprendan el idioma inglés.

#### 1.2.2. Objetivos específicos

- Crear un nuevo lenguaje de programación con una sintaxis similar a un pseudocódigo.
- Los token serán en español.
- Implementar el compilador del nuevo lenguaje de programación.
- Proveer un IDE para escribir y compilar programas escritos en el nuevo lenguaje de programación.

## **1.3. Alcance y limitaciones**

### **1.3.1. Resultados esperados**

Se espera que el nuevo lenguaje y compilador pueda ayudar a estudiantes con dificultad en el idioma inglés, así mismo ayuden a entender los programas más fácilmente y contar con una sintaxis que sea lo mas intuitiva posible. Finalmente facilitar la enseñanza de la programación de algoritmos dirigida a alumnos.

### **1.3.2. Delimitación del proyecto**

El lenguaje propuesto soportara las operaciones aritméticas básicas como la suma, resta, multiplicación y división. Así mismo las instrucciones básicas como el If, Else, For, While y la creacion de subrutinas o funciones. De modo que este lenguaje no se espera que se utilice para el desarrollo en general sino mas bien en un ámbito educativo para introducir a las personas en la programación de algoritmos. El IDE soportará las funciones básicas como abrir y guardar archivos, compilar y ejecutar programas escritos en el lenguaje de programación creado.

## **1.4. Justificación y viabilidad**

Crear un lenguaje de programación en español con una sintaxis similar a un pseudocódigo y un compilador y un IDE correspondiente podría ser una solución viable para abordar el problema de la mala comprensión del inglés en las clases de programación. Esto permitiría a los estudiantes centrarse en comprender los conceptos y a los profesores centrarse en la enseñanza sin tener que traducir términos técnicos. Aunque esto puede ser un proceso complejo y enfrentar resistencia al cambio, puede significar una mejora para la accesibilidad y la inclusión en la educación y en la formación de futuros programadores.

# Capítulo 2

## Marco teórico

### 2.1. Lenguajes de programación

Los lenguajes de programación son un conjunto de reglas, símbolos y convenciones que permiten a los programadores escribir programas informáticos. Estos lenguajes se clasifican según su nivel de abstracción y su paradigma de programación. [Sebesta \(2012\)](#)

### 2.2. Sintaxis de los lenguajes de programación

La sintaxis de un lenguaje de programación es el conjunto de reglas que establecen cómo se deben escribir las instrucciones de un programa. La sintaxis es fundamental para la comprensión de los programas y para su correcta ejecución por parte del computador. [Alfred V. Aho y Ullman \(2006\)](#)

### 2.3. Pseudocódigo

El pseudocódigo es un lenguaje de programación informal que se utiliza para describir algoritmos y programas de manera clara y sencilla. El pseudocódigo no tiene una sintaxis formal y no está diseñado para ser compilado o interpretado por un computador. [Joyanes Aguilar \(2015\)](#)

### 2.4. Enseñanza de la programación

La enseñanza de la programación se enfoca en el desarrollo de habilidades y conocimientos para la creación de software. Existen distintas metodologías de enseñanza, como la programación orientada a objetos, la programación estructurada, entre otras. [Kelleher y Pausch \(2005\)](#)

## 2.5. Compiladores

Un compilador es un programa informático que traduce el código fuente de un lenguaje de programación a un lenguaje que pueda ser ejecutado por el computador. El compilador es una herramienta fundamental para la creación de nuevos lenguajes de programación. [Muchnick \(1997\)](#)

## 2.6. Entornos de desarrollo integrados (IDE)

Un IDE es una herramienta que permite a los programadores escribir, depurar y ejecutar programas de manera integrada. Los IDEs suelen incluir un editor de código, un compilador, un depurador, entre otras herramientas.

## 2.7. Código Intermedio

es una representación de bajo nivel y sin dependencia de la plataforma utilizada en el proceso de compilación de un programa. Sirve como un puente entre el código fuente original y el código de máquina específico de una arquitectura. Proporciona portabilidad y permite realizar optimizaciones durante la compilación. Simplifica el desarrollo de compiladores al permitir etapas modulares y independientes. [Torczon \(2011\)](#)

## 2.8. Código abierto

Hace referencia al software cuyo código fuente está disponible para que cualquier persona lo estudie, modifique y redistribuya. Fomenta la colaboración, transparencia y libertad de uso del software. Ejemplos populares de proyectos de código abierto incluyen Linux, GCC y Firefox. [ORG \(2023\)](#)

## 2.9. LLVM (Low-Level Virtual Machine)

Es un conjunto de herramientas y bibliotecas de compilación. Utiliza una representación intermedia llamada "LLVM IR" para generar código eficiente e independiente de la plataforma. Incluye componentes como Clang (compilador de C/C++), optimizaciones de código y generación de código de máquina para diferentes arquitecturas. [Adve \(2015\)](#)

## 2.10. Método Shift-Reduce

Es una técnica utilizada en el análisis sintáctico de compiladores. Se basa en dos operaciones principales: "shift" (desplazamiento) y "reduce" (reducción). Durante el análisis, se toma un token de la entrada y se coloca en la pila (shift), o se aplican reglas gramaticales para reducir una secuencia

de símbolos en la pila a un símbolo no terminal (reduce). Estas operaciones se repiten hasta que se logra una estructura de árbol de derivación que representa la estructura gramatical del programa analizado. El método Shift-Reduce se utiliza en algoritmos de análisis sintáctico como el algoritmo LR y se implementa en herramientas como Yacc/Bison. [Alfred V. Aho y Ullman \(2006\)](#)

## 2.11. Tabla de Símbolos

La tabla de símbolos es una estructura de datos utilizada en compiladores y otros sistemas de procesamiento de lenguajes de programación. Esta tabla almacena información relevante sobre los símbolos encontrados en el código fuente, como variables, funciones y tipos de datos. Cada símbolo se representa mediante una entrada en la tabla que contiene detalles como su nombre, tipo, alcance y posición en la memoria. La tabla de símbolos es esencial para realizar análisis semántico, resolución de referencias y generación de código. [Fischer, LeBlanc, y Cytron \(2011\)](#)

## 2.12. Tabla de Códigos

La tabla de códigos es una estructura utilizada en compiladores y otros sistemas de traducción de lenguajes para asignar códigos o identificadores a elementos del lenguaje fuente. Estos códigos se utilizan posteriormente en el proceso de generación de código para representar de manera compacta las instrucciones o elementos del programa. Por ejemplo, en un compilador de lenguaje ensamblador, la tabla de códigos puede asignar códigos numéricos a las instrucciones y registros del procesador para facilitar la traducción del código fuente a instrucciones de máquina. [Fischer y cols. \(2011\)](#)



## Capítulo 3

# Estado del arte

### 3.1. DEFINICIÓN DEL LENGUAJE DE PROGRAMACIÓN EPLOAM PARA LA EJECUCIÓN DE PSEUDOCÓDIGO Y SU COMPILADOR

En este trabajo se presenta la enseñanza de algoritmos con pseudocódigo y la implementación de un lenguaje de alto nivel con una sintaxis similar al pseudocódigo utilizando Java y JavaCC. El lenguaje planteado se llama EPLOAM y sus gramática es en español. La ejecución del programa se hace desde terminal. [Arboleda \(2011\)](#)

### 3.2. Compilador y traductor de pseudocódigo para la lógica de programación (CompiProgramación)

En este trabajo se plantea una herramienta llamada CompiProgramación para la enseñanza de la lógica de programación, permite la creación de algoritmos usando pseudocódigo y finalmente la traducción de este a otros lenguajes como C++ o Java. [Vanegas \(2005\)](#)

### 3.3. COMPILADOR DE PSEUDOCÓDIGO COMO HERRAMIENTA PARA EL APRENDIZAJE EN LA CONSTRUCCIÓN DE ALGORITMOS

En este trabajo se plantea el uso de pseudocódigo para la enseñanza de la programación de algoritmos utilizando algunas herramientas como C++, Java, JFlex, PHP, HTML, JavaScript y CSS. Para crear un entrono gráfico en la web desde donde se puede ejecutar y utilizar el lenguaje similar al pseudocódigo. [Vega Castro \(2010\)](#)

### 3.4. HITO: Pseudocode compiler with graphics library

En este trabajo se plantea HITO como compilador para enseñar la lógica de programación . El primer modulo es la programación de algoritmos con pseudocódigo. Posteriormente la traducción de este lenguaje a otros como C++, Java y C#. Finalmente la creación de un entorno gráfico del cual se obtuvo las ideas para definir nuestro IDE. [Mamani Vilca, Muñoz Miranda, y Tumi Figueroa \(2019\)](#)

# Capítulo 4

## Metodología

### 4.1. Herramientas

Se usara las herramientas como Lex y Yacc que son un escáner o tokenizador y un parser correspondientemente. Esto nos ayudara con el trabajo de realizar el análisis sintáctico y semántico del código ingresado por el usuario. Se utilizara sus implementaciones en python con la librería `PLY` junto a la librería `Tkinter` para crear el IDE gráfico.

### 4.2. Diseño del lenguaje

En esta sección definiremos el diseño del nuevo lenguaje para lo cual nos basaremos en la guía de pseudocódigo del GCSE [Cambridge y RSA \(2015\)](#) con algunas variaciones donde lo más resaltante sera que las palabras reservadas estarán en español.

#### 4.2.1. Patrón de los tokens y palabras reservadas

A continuación definiremos los patrones de los tokens que se usaran dentro del lenguaje de programación.

Operadores aritméticos		
Token	Patrón	Lexemas
PLUS	símbolo +	+
MINUS	símbolo -	-
TIMES	símbolo *	*
DIVIDE	símbolo /	/
EQUALS	símbolo =	=
PERCENT	símbolo %	%

Tabla 4.1: Operadores aritméticos

Operadores de comparación		
Token	Patrón	Lexemas
LESS_THAN	símbolo <	<
LESS_EQUAL	símbolo <=	<=
GREATER_THAN	símbolo >	>
GREATER_EQUAL	símbolo >=	>=
EQUALITY	símbolo ==	==
NOT_EQUALITY	símbolo <>	<>

Tabla 4.2: Operadores de comparación

Agrupadores y separadores de argumentos y elementos		
Token	Patrón	Lexemas
LPAREN	símbolo (	(
RPAREN	símbolo )	)
LBRACKET	símbolo [	[
RBRACKET	símbolo ]	]
COMMA	símbolo ,	,

Tabla 4.3: Agrupadores y separadores de argumentos y elementos

Variables y salto de linea		
Token	Patrón	Lexemas
VAR	Letras y _ seguido de cero o mas letras, números y _	num var1 var2
NEWLINE	Símbolo \n seguidos	\n \n\n

Tabla 4.4: Variables y salto de linea

Constantes		
Token	Patrón	Lexemas
DOUBLE_CONST	Uno o mas dígitos seguidos de un . seguido de cero o mas dígitos	0.5 45.84 6.
INT_CONST	Uno o mas dígitos seguidos	12 3 234 8764
STRING_CONST	Símbolo " seguido de cualquier cosa hasta finalizar con un "	" " "cadena"

Tabla 4.5: Constantes

Palabras reservadas		
Token	Patrón	Lexemas
INT	ENTERO	ENTERO
DOUBLE	DECIMAL	DECIMAL
INPUT	ENTRADA	ENTRADA
OUTPUT	SALIDA	SALIDA
SUBROUTINE	SUBROUTINA	SUBROUTINA
ENDSUBROUTINE	FINSUBROUTINA	FINSUBROUTINA
RETURN	DEVOLVER	DEVOLVER
IF	SI	SI
THEN	ENTONCES	ENTONCES
ELSE	SINO	SINO
ENDIF	FINSI	FINSI
WHILE	MIENTRAS	MIENTRAS
DO	HACER	HACER
ENDWHILE	FINMIENTRAS	FINMIENTRAS
FOR	PARA	PARA
TO	HASTA	HASTA
ENDFOR	FINPARA	FINPARA

Tabla 4.6: Palabras reservadas

#### 4.2.2. Definición de la gramática

En esta sección definiremos la gramática y sus reglas para definir la sintaxis del lenguaje de programación. Estas reglas se definirán como sigue:

- Instrucción básica del programa

```
stmt_list: simple_stmt
          | stmt_list simple_stmt;
```

```
simple_stmt: assig_stmt
           | array_decl_stmt
           | var_decl_stmt
           | if_stmt
           | while_stmt
           | for_stmt
```

```

        | output_stmt
        | input_stmt
        | function_stmt
        | return_stmt;

```

- Instrucción if

```

if_stmt: IF exp THEN stmt_list ENDIF
        | IF exp THEN stmt_list ELSE stmt_list ENDIF;

```

- Instrucción while

```

while_stmt: WHILE exp DO stmt_list ENDWHILE;

```

- Instrucción for

```

for_stmt: FOR assig_stmt TO INTCONST stmt_list ENDFOR;

```

- Instrucciones de declaración

```

array_decl_stmt: DOUBLE array_index
                | INT array_index;

```

```

var_decl_stmt: DOUBLE var_list
              | INT var_list;

```

```

var_list: VAR
         | var_list COMMA VAR;

```

- Instrucciones de asignación

```

assig_stmt: VAR EQUALS exp { $$ = $3; }
          | array_index EQUALS exp;
array_index: VAR LBRACKET INTCONST RBRACKET;

```

- Instrucciones de salida y entrada de datos

```

input_stmt: INPUT VAR
           | INPUT array_index;
output_stmt: OUTPUT exp { printf("VALOR -> %g\n", $2); };
return_stmt: RETURN exp;

```

- Instrucción de funciones

```

function_header: INT SUBROUTINE VAR LPAREN arg_list RPAREN
                | DOUBLE SUBROUTINE VAR LPAREN arg_list RPAREN
                | INT SUBROUTINE VAR LPAREN RPAREN
                | DOUBLE SUBROUTINE VAR LPAREN RPAREN;

```

```

arg_list: INT VAR
         | DOUBLE VAR
         | arg_list COMMA INT VAR
         | arg_list COMMA DOUBLE VAR;

```

```

function_stmt: function_header stmt_list ENDSUBROUTINE;

```

```

exp_fun_call: VAR LPAREN exp RPAREN
             | VAR LPAREN expr_list RPAREN
             | VAR LPAREN RPAREN;

```

```

expr_list: exp COMMA exp
         | expr_list COMMA exp;

```

■ Instrucciones de expresiones aritméticas y de comparación

```

exp: exp EQUALITY term
    | exp NOTEQUALITY term
    | term { $$ = $1; };

```

```

term: term LESSTHAN exp_arit
     | term GREATERTHAN exp_arit
     | term LESSEQUAL exp_arit
     | term GREATEREQUAL exp_arit
     | exp_arit { $$ = $1; };

```

```

exp_arit: exp_arit PLUS term_arit1 { $$ = $1 + $3; }
        | exp_arit MINUS term_arit1 { $$ = $1 - $3; }
        | term_arit1 { $$ = $1; };

```

```

term_arit1: term_arit1 TIMES term_arit2 { $$ = $1 * $3; }
          | term_arit1 DIVIDE term_arit2 { $$ = $1 / $3; }
          | term_arit1 PERCENT term_arit2 { $$ = (int)$1 % (int)$3; }
          | term_arit2 { $$ = $1; };

```

```

term_arit2: MINUS term_arit2 %prec UMINUS { $$ = -$2; }
          | term_arit3 { $$ = $1; };

```

```

term_arit3: LPAREN exp RPAREN { $$ = $2; }
          | INTCONST

```



```
| DOUBLECONST
| STRINGCONST
| VAR
| array_index
| exp_fun_call;
```

### 4.3. Implementación del lenguaje

Para la implementación del compilador se usara el lenguaje de programación Python y las herramientas Lex y Yacc. Usaremos una implementación completamente en Python de estas herramientas con la librería de PLY. Junto a TK para crear el entorno grafico IDE.

#### 4.3.1. Implementación de los tokens con Lex y Python

Para definir los tokens con Lex se usan expresiones regulares las cuáles definiremos siguiendo los patrones establecidos anteriormente en la sección de diseño del lenguaje.

#### 4.3.2. Implementación de la gramática con Yacc y Python

Para la implementación de la gramática seguiremos las reglas del lenguaje definidas anteriormente en la sección de diseño del lenguaje. Así mismo se definirán los pasos y acciones a seguir con los tokens que lleguen en cada regla, realizando una validación de tipos de dato de cada variable así como las operaciones entre variables según el tipo de dato y verificando una previa definición de la variable antes de su uso. Finalmente se finalizara con la obtención de un árbol de sintaxis abstracta.

#### 4.3.3. Implementación de nodos básicos del AST

La implementación de los nodos básicos se realizara de tal forma que se almacene la información importante de cada bloque básico del lenguaje definido. Esta información se utilizara posteriormente durante la compilación.

#### 4.3.4. Implementación del generador de código intermedio

La implementación del generador de código de intermedio se realizara de modo que realizara un recorrido por los nodos del árbol de sintaxis abstracta utilizando la información almacenada previamente para realizar las instrucciones necesarias con el lenguaje intermedio.

## 4.4. Diseño del IDE

Tomando como referencia algunos editores y considerando solo las funcionalidades básicas para el uso del lenguaje de programación tales como editar, guardar y abrir archivos. Las funciones compilar y ejecutar el programa se realizarán utilizando las implementaciones del tokenizador y parser realizados anteriormente.

## 4.5. Implementación del IDE

Para la implementación de IDE se utilizará el lenguaje de programación Python y la librería Tk para la creación de la caja de texto de entrada, opciones del menú y resaltado sintáctico.



```

Spanish Pseudocode Compiler - C:/Users/USUARIO/Documents/Github/Pseudocode-Compiler/examples/code.spl
File
# declaracion de variables
ENTERO x, y, z
DECIMAL arr[3]

x = 12 - 3
arr[2] = 23.7

ENTERO SUBROUTINA func(ENTERO a, DECIMAL b)

a = 1 * 4 + (4 / 2)

DEVOLVER a + 2

FINSUBROUTINA

y = func(12, 0.5)

SI x <= 0 ENTONCES
    y = x + 12
    DEVOLVER y
SINO
    z = 2 * x
    DEVOLVER z
FINSI

MIENTRAS y <> 100 HACER
    ENTRADA y
    SALIDA fibonacci(y)
FINMIENTRAS

PARA x = 0 HASTA 7
    y = y + x
FINPARA

SALIDA "cadena"
  
```

Figura 4.1: Interfaz gráfica (IDE)

## 4.6. Nombre del compilador

El lenguaje definido será una mezcla de la sintaxis de pseudocódigo pero en español, para así facilitar su entendimiento, por tal motivo el nombre asignado al compilador será Spanish Pseudocode language (SPL).

## Capítulo 5

# Experimentos y resultados

### 5.1. Procedimiento

Para realizar el proceso de compilación del lenguaje, el flujo que se lleva a cabo es el siguiente:

- Creación de un escáner o tokenizador con Lex para definir el patrón de cada token.
- Al pasar el código fuente por el tokenizador obtendremos una arreglo de token el cual pasara por un análisis sintáctico.
- Creación de la gramática que definirá la sintaxis del lenguaje que se creará con Yacc. Después de que el arreglo de tokens pase por este, finalmente obtendremos el código de maquina.

### 5.2. Pruebas

En base a la gramática definida realizamos las siguientes pruebas donde se pudo verificar que reconoce correctamente los patrones de tokens establecidos.

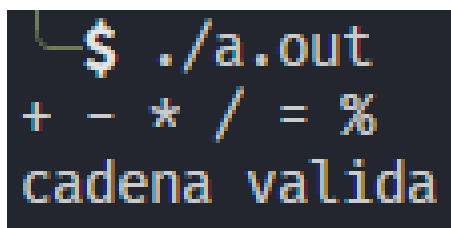


Figura 5.1: Operadores aritméticos

```

└─ ⌚ 19s885ms $ ./a.out
< <= > >= == <>
cadena valida

```

Figura 5.2: Operadores de comparación

```

└─ ⌚ 19s811ms $ ./a.out
( ) [ ] ,
cadena valida

```

Figura 5.3: Agrupadores y separador de argumentos y elementos

```

└─ ⌚ 9s558ms $ ./a.out
num var1 var2

cadena valida

```

Figura 5.4: Variables y salto de línea

```

└─ ⌚ 33s697ms $ ./a.out
0.5 3. 34.35
12 25
cadena valida

```

Figura 5.5: Constantes

```

└─ 🕒 1m20s756ms $ ./a.out
ENTERO
DECIMAL
ENTRADA SALIDA
SUBROUTINA FINSUBROUTINA
DEVOLVER
SI ENTONCES SINO
FINSI
MIENTRAS HACER FINMIENTRAS
PARA HASTA FINPARA
cadena valida

```

Figura 5.6: Palabras reservadas

Para probar las operaciones aritméticas se han definido los cálculos que se realizarán en cada reducción de las expresiones aritméticas y el de salida de datos para realizar las siguientes pruebas:

- Entrada  $2 + 3 * 2 - 8 / 2$

```

SALIDA 2+3*2-8/2
VALOR -> 4

```

- Entrada  $20 / 2 + 20 * 1 - 10$

```

SALIDA 20/2+20*1-10
VALOR -> 20

```

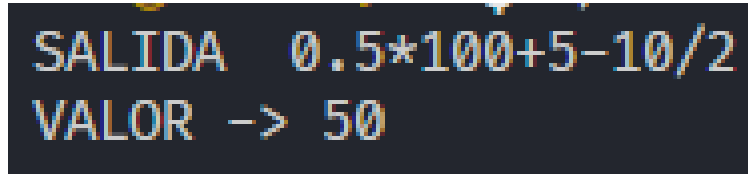
- Entrada  $300 * 3 + 30 * 3 + 2 * 3 + 2 - 1$

```

SALIDA 300*3+30*3+2*3+2-1
VALOR -> 997

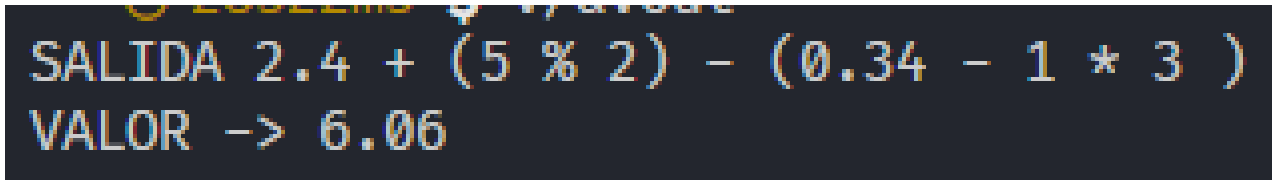
```

- Entrada  $0.5 * 100 + 5 - 10/2$



```
SALIDA 0.5*100+5-10/2
VALOR -> 50
```

- Entrada  $2.4 + (5 \% 2) - (0.34 - 1 * 3)$



```
SALIDA 2.4 + (5 % 2) - (0.34 - 1 * 3 )
VALOR -> 6.06
```

### 5.2.1. Árboles de sintaxis abstracta

A continuación se mostrara los arboles sintácticos de las entradas de prueba.

Definimos una primera estructura general para los arboles de las pruebas.

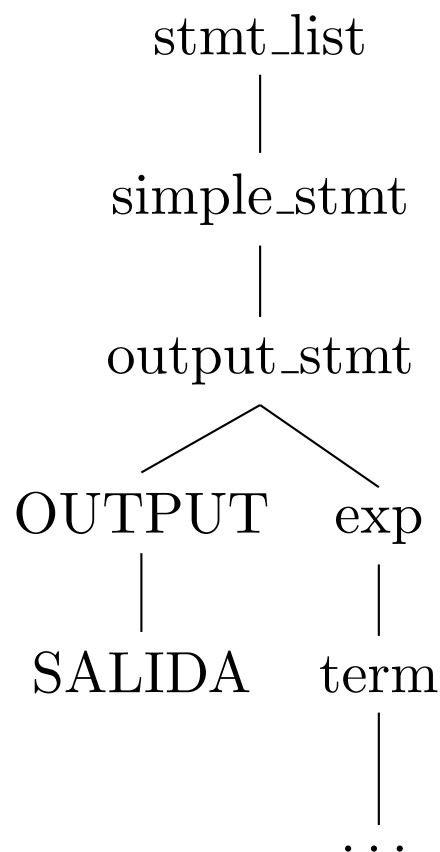
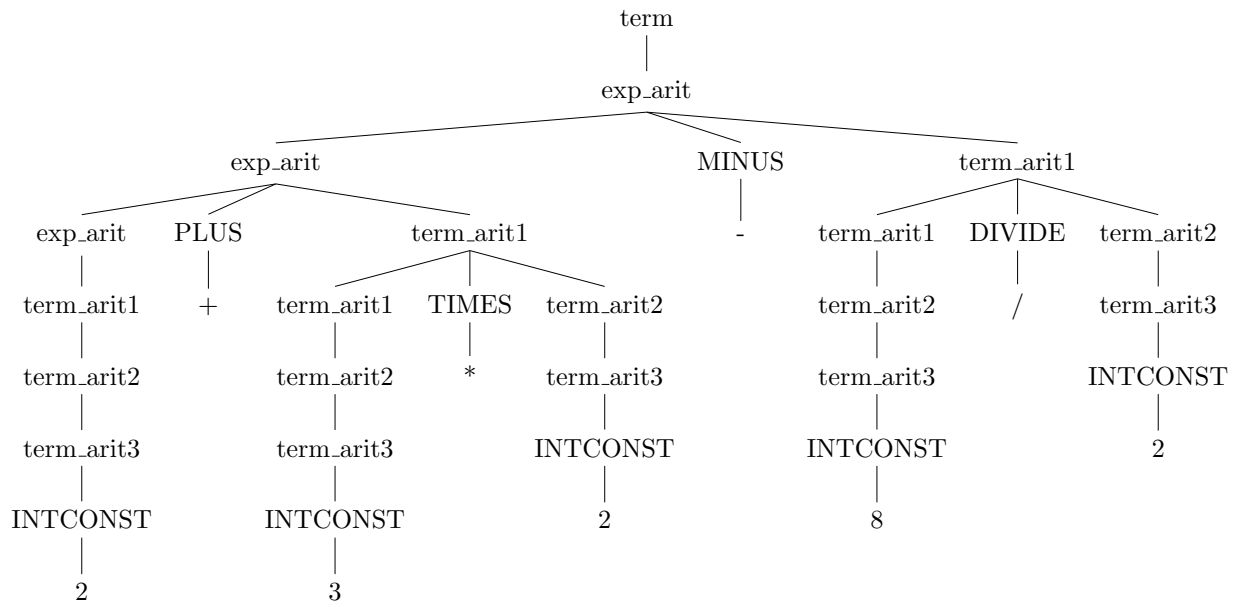
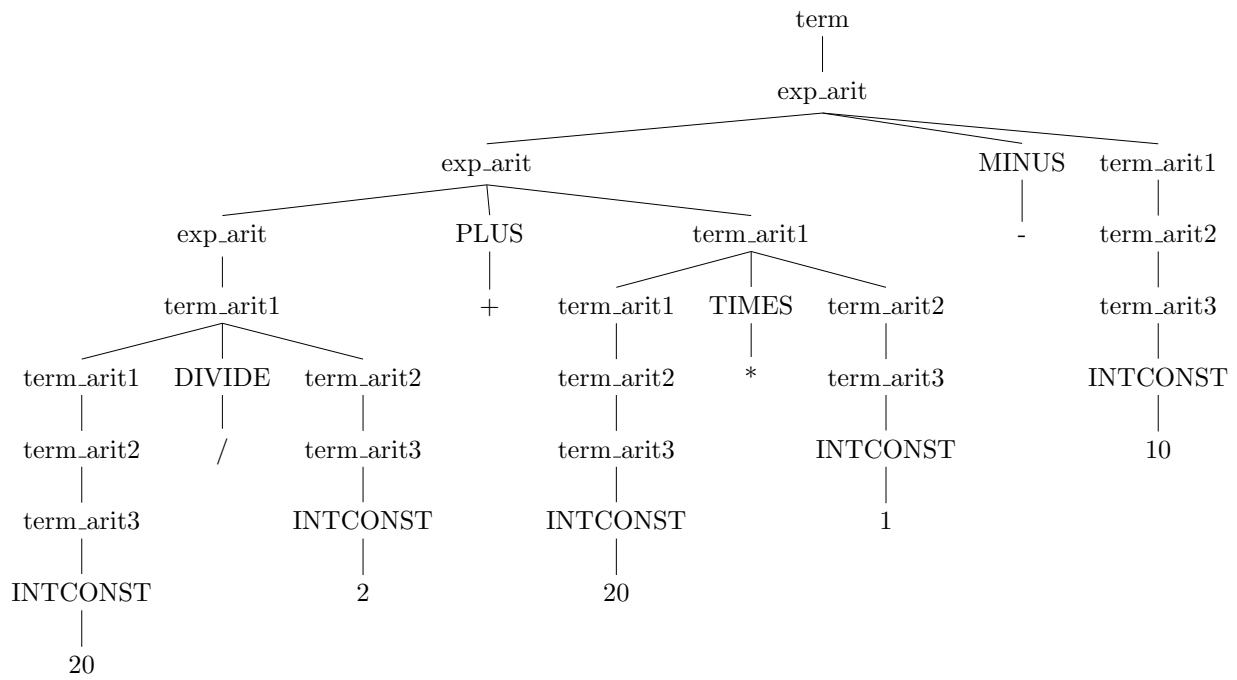


Figura 5.7: Entrada SALIDA ...

Figura 5.8: Entrada  $2 + 3 * 2 - 8 / 2$ Figura 5.9: Entrada  $20/2 + 20 * 1 - 10$

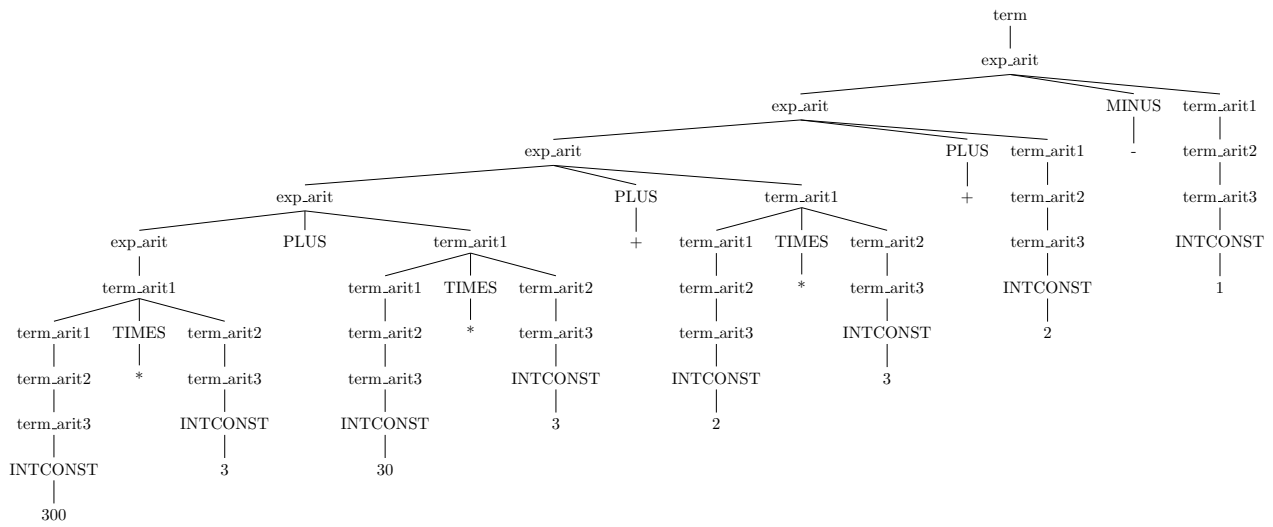


Figura 5.10: Entrada  $300 * 3 + 30 * 3 + 2 * 3 + 2 - 1$

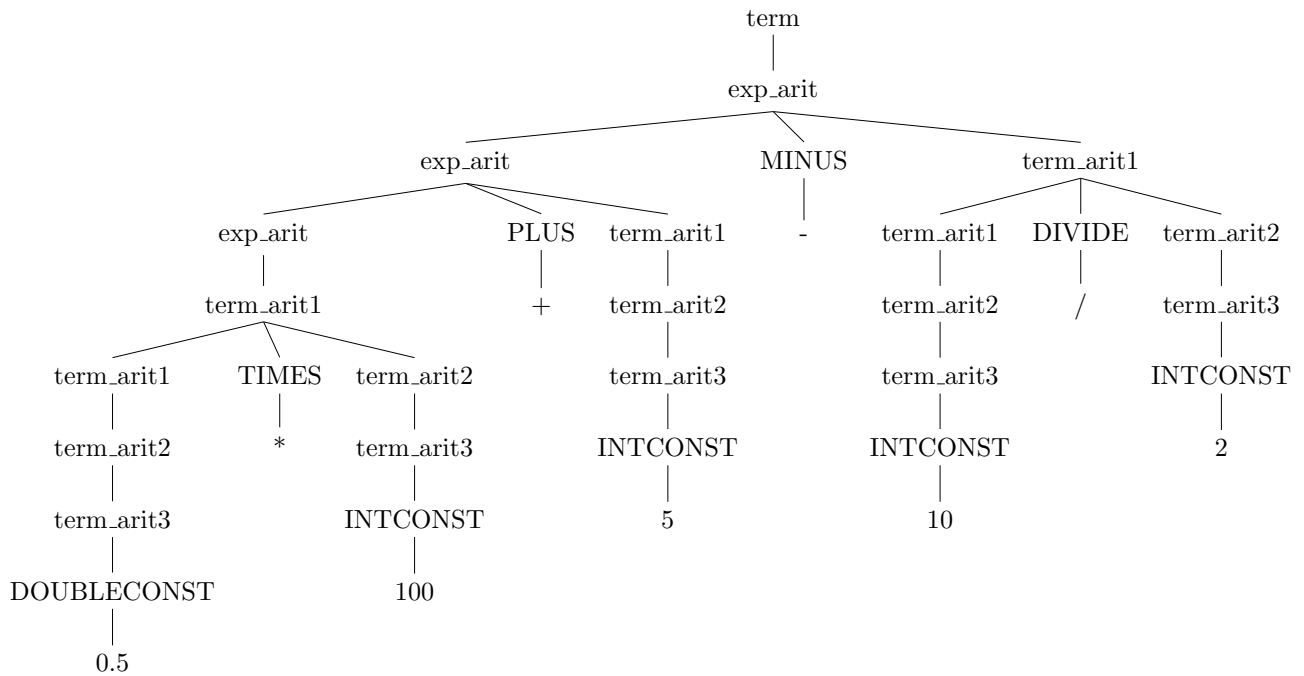


Figura 5.11: Entrada  $0.5 * 100 + 5 - 10/2$



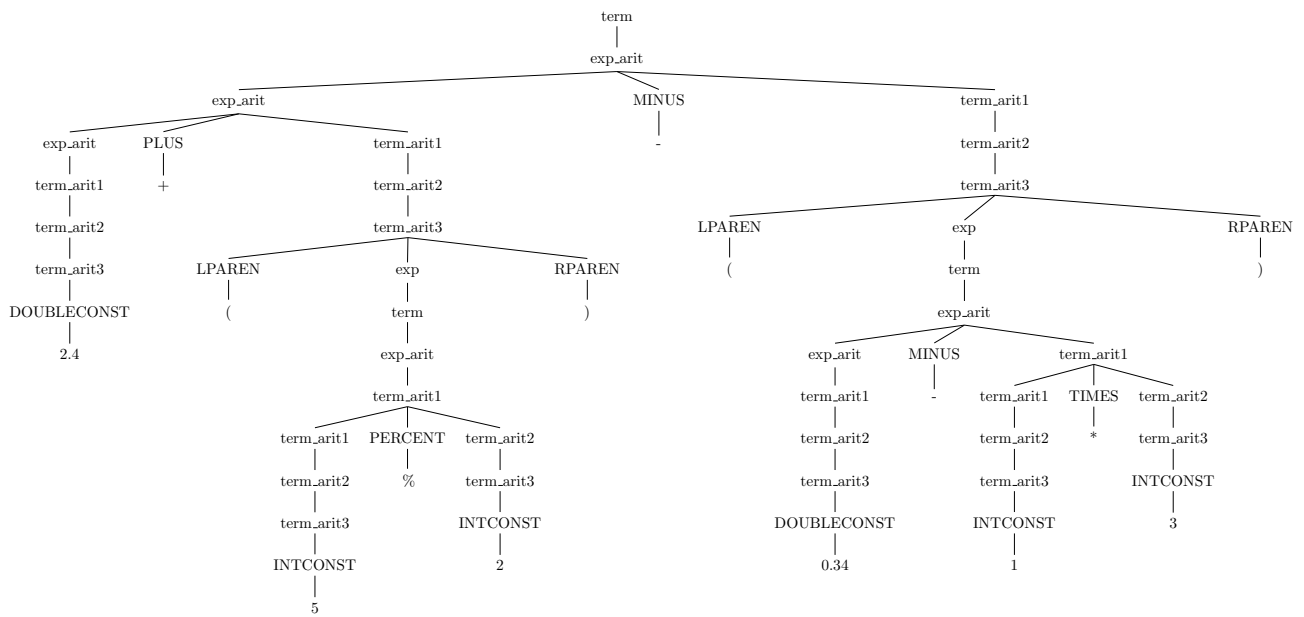


Figura 5.12: Entrada  $2.4 + (5 \% 2) - (0.34 - 1 * 3)$

### 5.2.2. Método Shift/Reduce para verificar la validez de los caracteres de entrada

El método consiste en leer token por token siguiendo los pasos y reglas establecidas en cada estado hasta llegar un estado de aceptación.

Estados	Acción	Cadena
		SALIDA 2 + 3
0	OUTPUT shift, and go to state 5	OUTPUT INTCONST PLUS INTCONST
0 5	INTCONST shift, and go to state 36	INTCONST PLUS INTCONST
0 5 36	default reduce using rule 62 (term_arit3)	PLUS INTCONST
0 5	term_arit3 go to state 48	PLUS INTCONST
0 5 48	default reduce using rule 60 (term_arit2)	PLUS INTCONST
0 5	term_arit2 go to state 47	PLUS INTCONST
0 5 47	default reduce using rule 58 (term_arit1)	PLUS INTCONST
0 5	term_arit1 go to state 46	PLUS INTCONST
0 5 46	default reduce using rule 54 (exp_arit)	PLUS INTCONST
0 5	exp_arit go to state 45	PLUS INTCONST
0 5 45	PLUS shift, and go to state 71	PLUS INTCONST
0 5 45 71	INTCONST shift, and go to state 36	INTCONST
0 5 45 71 36	default reduce using rule 62 (term_arit3)	\$
0 5 45 71	term_arit3 go to state 48	\$
0 5 45 71 48	default reduce using rule 60 (term_arit2)	\$
0 5 45 71	term_arit2 go to state 47	\$
0 5 45 71 47	default reduce using rule 58 (term_arit1)	\$
0 5 45 71	term_arit1 go to state 95	\$
0 5 45 71 95	default reduce using rule 52 (exp_arit)	\$
0 5	exp_arit go to state 45	\$
0 5 45	default reduce using rule 51 (term)	\$
0 5	term go to state 44	\$
0 5 44	default reduce using rule 46 (exp)	\$
0 5	exp go to state 43	\$
0 5 43	default reduce using rule 27 (output_stmt)	\$
0	output_stmt go to state 19	\$
0 19	default reduce using rule 9 (simple_stmt)	\$
0	simple_stmt go to state 11	\$
0 11	default reduce using rule 1 (stmt_list)	\$
0	stmt_list go to state 10	\$
0 10	end shift, and go to state 53	\$
0 10 53	default accept	accept

Tabla 5.1: Metodo Shift/Reduce para SALIDA 2 + 3

### 5.2.3. Tabla de Símbolos

Estructura de datos que almacena información sobre los símbolos en el código fuente, como variables y funciones. Ayuda a realizar análisis semántico y generación de código.

- Entrada

```
x = 2.5
y = 3 + 4

SALIDA x + y / 2 * 3
SALIDA z * 2 / 7 - 6 + 4

z = x * 3 / y + 45
```

- Tabla de símbolos

```
tabla de simbolos
0  nombre=x tok=258 valor=0.000000
1  nombre=2.5 tok=260 valor=2.500000
2  nombre=y tok=258 valor=0.000000
3  nombre=3 tok=259 valor=3.000000
4  nombre=4 tok=259 valor=4.000000
5  nombre=_T0 tok=258 valor=0.000000
6  nombre=2 tok=259 valor=2.000000
7  nombre=_T1 tok=258 valor=0.000000
8  nombre=_T2 tok=258 valor=0.000000
9  nombre=_T3 tok=258 valor=0.000000
10 nombre=z tok=258 valor=0.000000
11 nombre=_T4 tok=258 valor=0.000000
12 nombre=7 tok=259 valor=7.000000
13 nombre=_T5 tok=258 valor=0.000000
14 nombre=6 tok=259 valor=6.000000
15 nombre=_T6 tok=258 valor=0.000000
16 nombre=_T7 tok=258 valor=0.000000
17 nombre=_T8 tok=258 valor=0.000000
18 nombre=_T9 tok=258 valor=0.000000
19 nombre=45 tok=259 valor=45.000000
20 nombre=_T10 tok=258 valor=0.000000
```

### 5.2.4. Tabla de Códigos

Estructura que asigna códigos o identificadores a elementos del lenguaje fuente. Se utiliza para representar de forma compacta instrucciones o elementos del programa durante la generación de código.

- Entrada

```
x = 2.5
y = 3 + 4

SALIDA x + y / 2 * 3
SALIDA z * 2 / 7 - 6 + 4

z = x * 3 / y + 45
```

- Tabla de códigos

```
tabla de codigos
op=297  a1=0 a2=1 a3=45
op=298  a1=5 a2=3 a3=4
op=297  a1=2 a2=5 a3=45
op=301  a1=7 a2=2 a3=6
op=300  a1=8 a2=7 a3=3
op=298  a1=9 a2=0 a3=8
op=300  a1=11 a2=10 a3=6
op=301  a1=13 a2=11 a3=12
op=299  a1=15 a2=13 a3=14
op=298  a1=16 a2=15 a3=4
op=300  a1=17 a2=0 a3=3
op=301  a1=18 a2=17 a3=2
op=298  a1=20 a2=18 a3=19
op=297  a1=10 a2=20 a3=45
```

## 5.3. Resultados

## 5.4. Análisis

### 5.4.1. Características

#### Lenguaje en español

El lenguaje de programación creado utiliza palabras clave y estructuras de programación en español. Esto facilita la comprensión y el aprendizaje para aquellos que no tienen un buen dominio del inglés.

#### Sintaxis similar a pseudocódigo

La sintaxis del lenguaje se asemeja al pseudocódigo, lo que lo hace más intuitivo y fácil de entender. Esto permite a los estudiantes y profesores enfocarse en la lógica y los conceptos de programación sin preocuparse por la sintaxis rigurosa de otros lenguajes.

### 5.4.2. Ventajas

#### Sintaxis fácil de entender

La sintaxis similar a pseudocódigo y en español facilita la comprensión de los programas. Los estudiantes pueden centrarse en la lógica y la estructura de los algoritmos sin tener que preocuparse por las complejidades sintácticas de otros lenguajes de programación.

#### Enseñanza de lógica de programación y algoritmos

El lenguaje de programación y el entorno de desarrollo asociado están diseñados específicamente para facilitar la enseñanza de la lógica de programación y los conceptos de algoritmos. Esto permite a los profesores y estudiantes enfocarse en los fundamentos de la programación sin tener que preocuparse por detalles técnicos innecesarios.

#### Accesibilidad para estudiantes con dificultades en inglés

Al proporcionar una solución en español, se aborda las barreras lingüísticas que pueden dificultar la enseñanza de programación a aquellos que no comprenden bien el inglés. Esto hace que la programación sea más accesible e inclusiva para un grupo más amplio de estudiantes.

#### Intuitivo y orientado a principiantes

Al centrarse en la enseñanza de algoritmos y conceptos básicos, el proyecto está diseñado para ser una introducción amigable a la programación. Esto proporciona a los principiantes una base sólida

para desarrollar habilidades de programación más avanzadas en el futuro.

# Referencias

- Adve, C. L. . V. (2015). *The llvm compiler infrastructure*. <https://www.cs.cornell.edu/~asampson/blog/llvm.html>. ([Online; accessed 9-jun-2023])
- Alfred V. Aho, R. S., Monica S. Lam, y Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools* (2nd ed.). Pearson Education.
- Arboleda, O. (2011). Definición del lenguaje de programación eploam para la ejecución de pseudocódigo y su compilador. *Scientia et technica*, 2(48), 116–121.
- Cambridge, O., y RSA. (2015). *Pseudocode guide*. <https://www.ocr.org.uk/images/202654-pseudocode-guide.pdf>. ([Online; accessed 4-may-2023])
- Fischer, C., LeBlanc, R., y Cytron, R. (2011). *Crafting a compiler*. Pearson Education. Descargado de <https://books.google.com.pe/books?id=GSYrAAAAQBAJ>
- Joyanes Aguilar, L. (2015). *Fundamentos de programación* (4th ed.). McGraw Hill.
- Kelleher, C., y Pausch, R. (2005, jun). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2), 83–137. Descargado de <https://doi.org/10.1145/1089733.1089734> doi: 10.1145/1089733.1089734
- Mamani Vilca, E., Muñoz Miranda, J. C., y Tumi Figueroa, E. N. (2019). Hito: Pseudocode compiler with graphics library. *Repositorio institucional UNAMBA*.
- Muchnick, S. S. (1997). *Advanced compiler design and implementation* (1st ed.). Morgan Kaufmann.
- ORG, O. (2023). *Código abierto*. <https://opensource.org/>. ([Online; accessed 9-jun-2023])
- Sebesta, R. W. (2012). *Concepts of programming languages* (11th ed.). Pearson Education.
- Torczon, K. D. C. . L. (2011). *Engineering a compiler* (2da ed.). Morgan Kaufmann.
- Vanegas, C. A. (2005). Compilador y traductor de pseudocódigo para la lógica de programación (compiprogramación). *Tecnura*, 8(16), 64–72.
- Vega Castro, R. A. (2010). *Compilador de pseudocódigo como herramienta para el aprendizaje en la construcción de algoritmos* (Tesis de Master no publicada). Pregrado en Ingeniería de Sistemas y Computación.