LENGUAJES Y TRADUCTORES
Proyecto

Segunda entrega
Diagramas de sintaxis del lenguaje completo

Profesora:
Norma Frida Roffe Samaniego

Alumno:
Emérico Pedraza Gómez

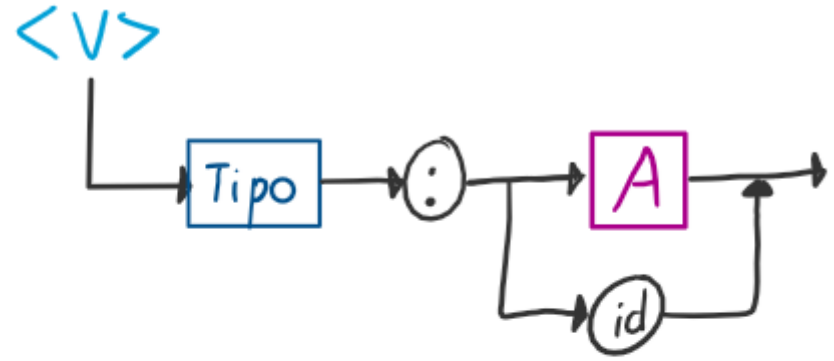
Matrícula:
A01382216

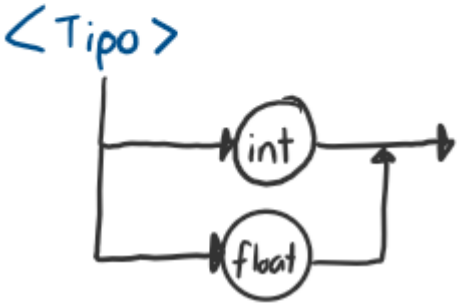
Fecha:
04 de abril de 2022

Nombre del lenguaje: M Pro

1. La sintaxis del lenguaje no es libre. Deberás tomar como base la sintaxis del lenguaje de programación ADA.
    a. El estándar que usaremos para el desarrollo de nuestro compilador es ADA 95. Lo localizas en
        i. http://www.gedlc.ulpgc.es/docencia/mp_i/GuiaAda/
        ii. El lenguaje es muy amplio, sólo toma en cuenta el subconjunto que aquí se solicita.
2. Dos tipos de datos numéricos:
    a. Entero y Flotante
        i. Para definir una variable entera
            1. int : x;
            2. int : x => 5;
        ii. Para definir una variable flotante
            1. float : y;
            2. float : y => 4.5;

Primer acercamiento a la definición de variables (incompleto):

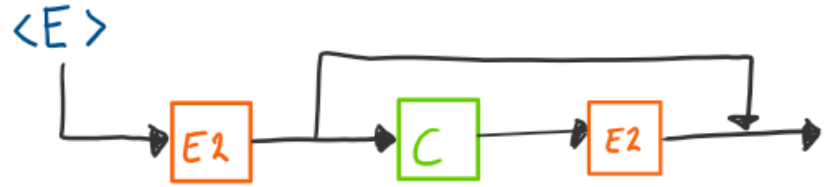


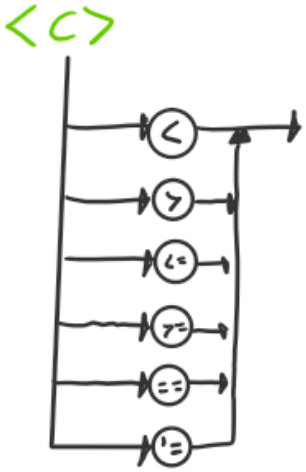


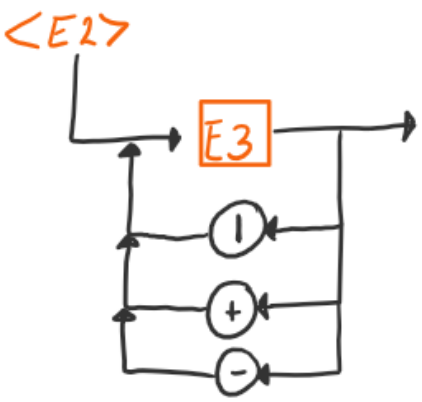
3. Operadores aritméticos básicos: +, -, *, /
   He decidido utilizar estos mismos operadores.
    a. Suma → +
    b. Resta → -
    c. Multiplicación → *
    d. División → /
    e. Módulo → %

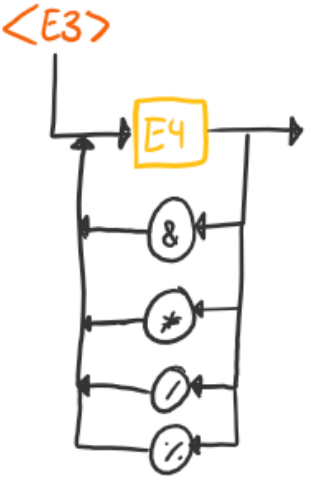
4. Operadores lógicos y relacionales básicos: AND, OR, NOT, >, <, <=, >=, <>, ==.
    a. AND → &
    b. OR → |
    c. NOT → ‘
    d. > → >
    e. < → <
    f. <= → >=
    g. >= → <=
    h. <> → ‘=
    i. == → ==

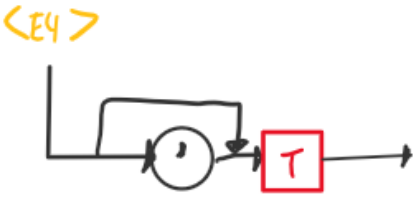
Expresiones combinadas:

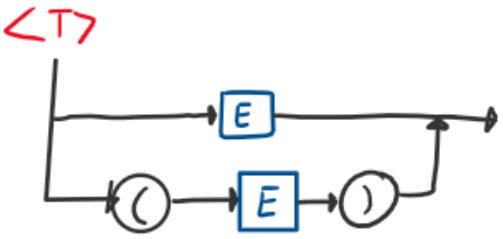






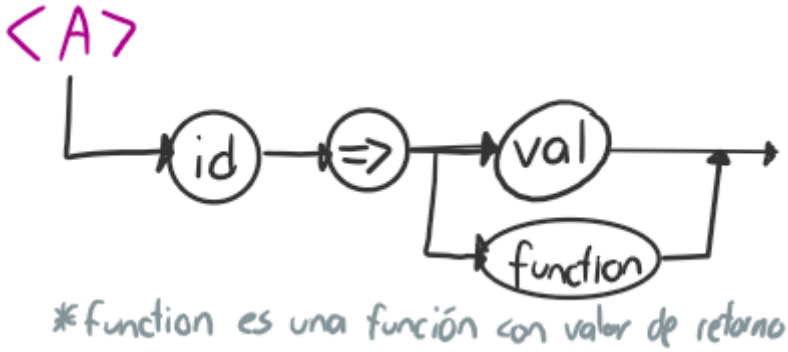






5. Operación de Asignación
    =>
    Por ejemplo:
        a **=>** 5;

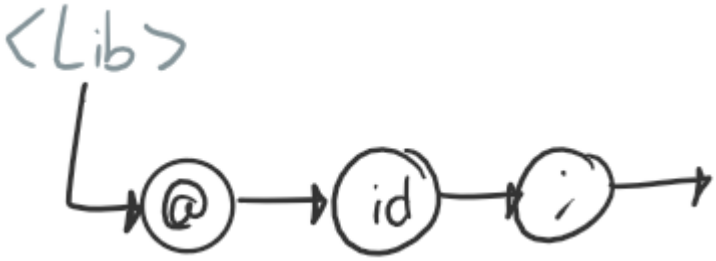
Asignación



6. Deberá soportar programación modular con procedimientos y funciones. Estructura básica de un programa con módulos (lo que le sigue al carácter '~' es un comentario), utilizar punto y coma siempre después de cada instrucción:

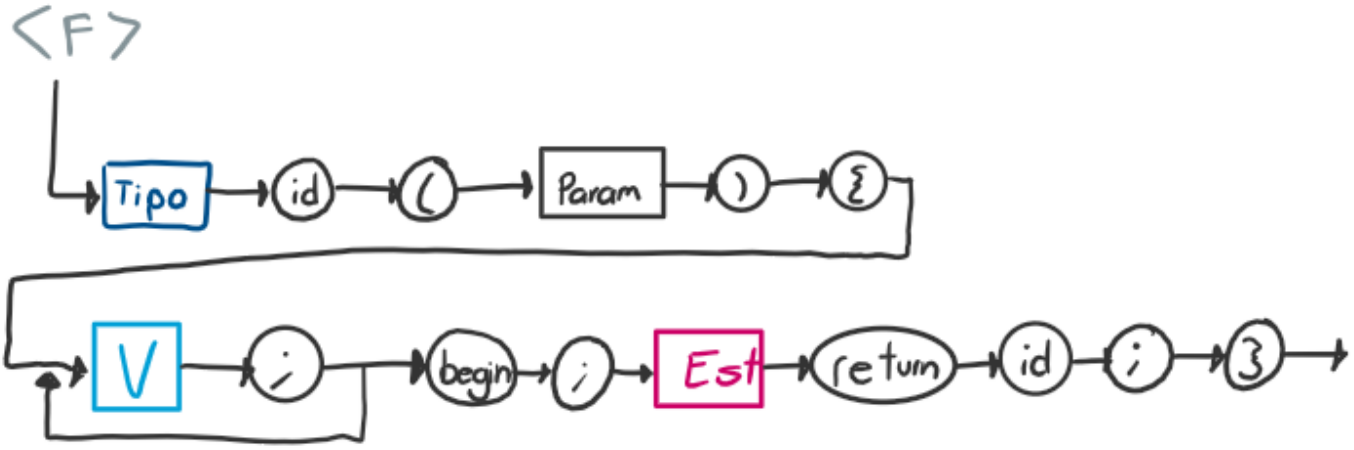
   @myLibrary; ~Uso de una librería

Librerías:



~ Estructura básica de una función.
float myModule**(**float param1, float param2**) {**
    float : result; ~ Definición de variable.
begin;
    result **=>** param1 + param2;
    return result; ~ Retorno de valor tipo float.
}

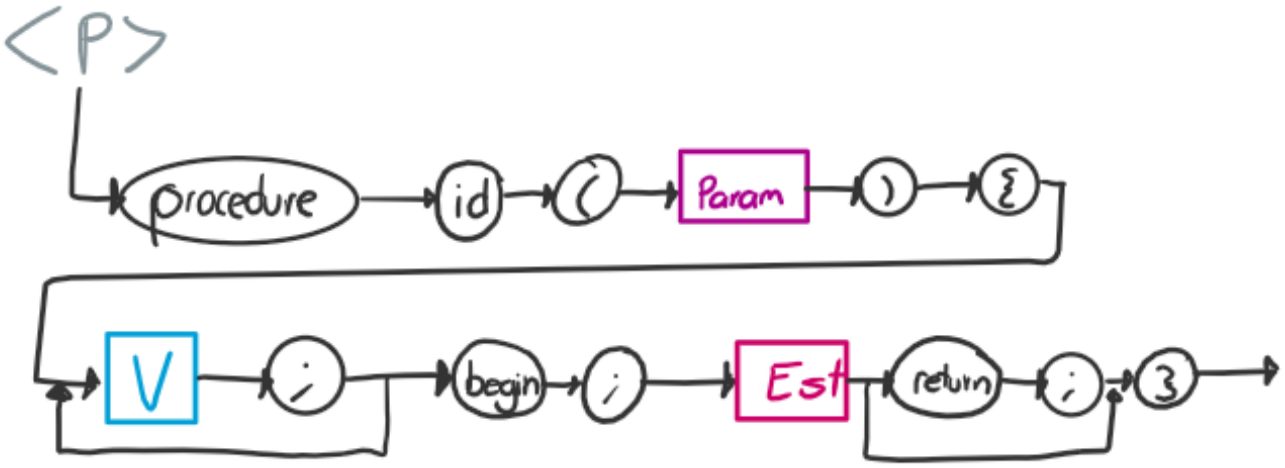
Función:



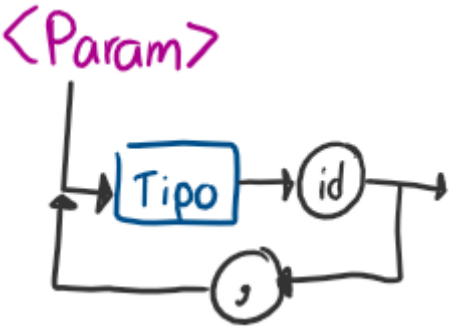
~ Estructura básica de un procedimiento.

```
procedure main( ) {
        float : varA;          ~ Definición de variable.
begin;
        print(2+2);
        return;                ~ No cuenta con valor de retorno, return opcional
}
```

Procedimiento:

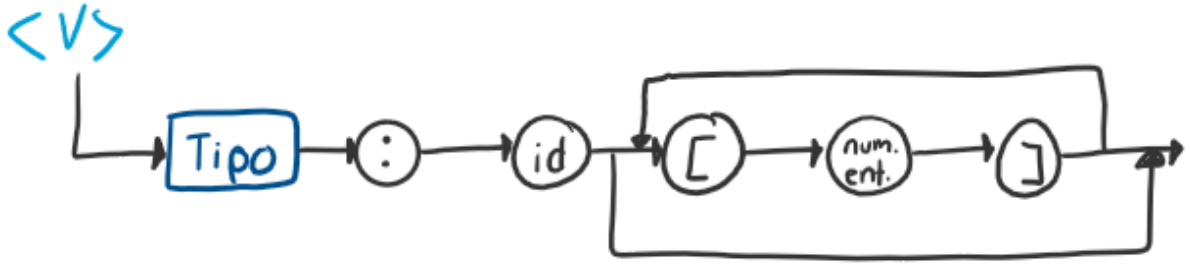


Parámetros:



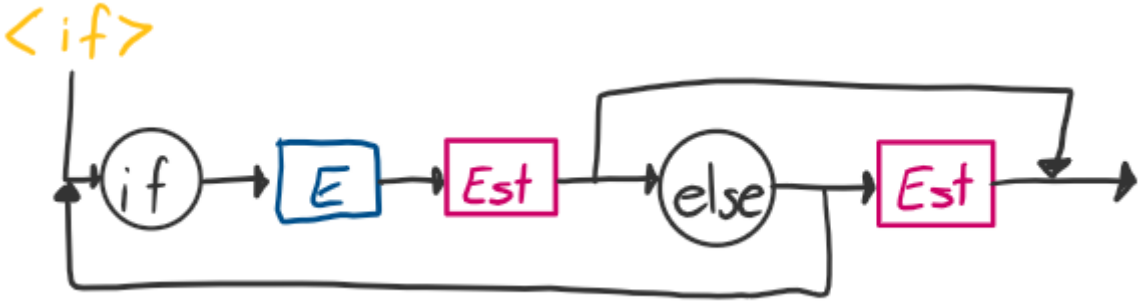
7. La implementación de variables dimensionadas es una parte fundamental de su proyecto, y no podrá ser omitida, se deben permitir hasta tres dimensiones. El tamaño de las dimensiones debe ser definido con constantes enteras.
   Utilizar **corchetes** para variables dimensionadas.
   a. Definición de una variable dimensionada:
      i. int : nums**[10]**;
      ii. int : nums3D**[5][5][3]**;
   b. Acceso a un elemento de una variable dimensionada:
      i. x => nums**[9]**;
      ii. y => nums**[2][2][2]**;

Definición de variables dimensionadas (completo):



8. Variables Globales. Contempla en tu proyecto el uso de Variables Locales para la sintaxis, su implementación (su ejecución) quedará opcional, será posible administrarlas como variables globales.
   Las variables locales se definen dentro de un módulo, mientras que las variables globales se definen fuera de todos los módulos, antes del primer módulo.
9. Instrucciones equivalentes (basadas en la sintaxis de ADA) a:
   a. **IF**

```
if A > B & A > C {
        ~ Estatutos
} else if A < C {
        ~ Estatutos
} else {
        ~ Estatutos
}
```

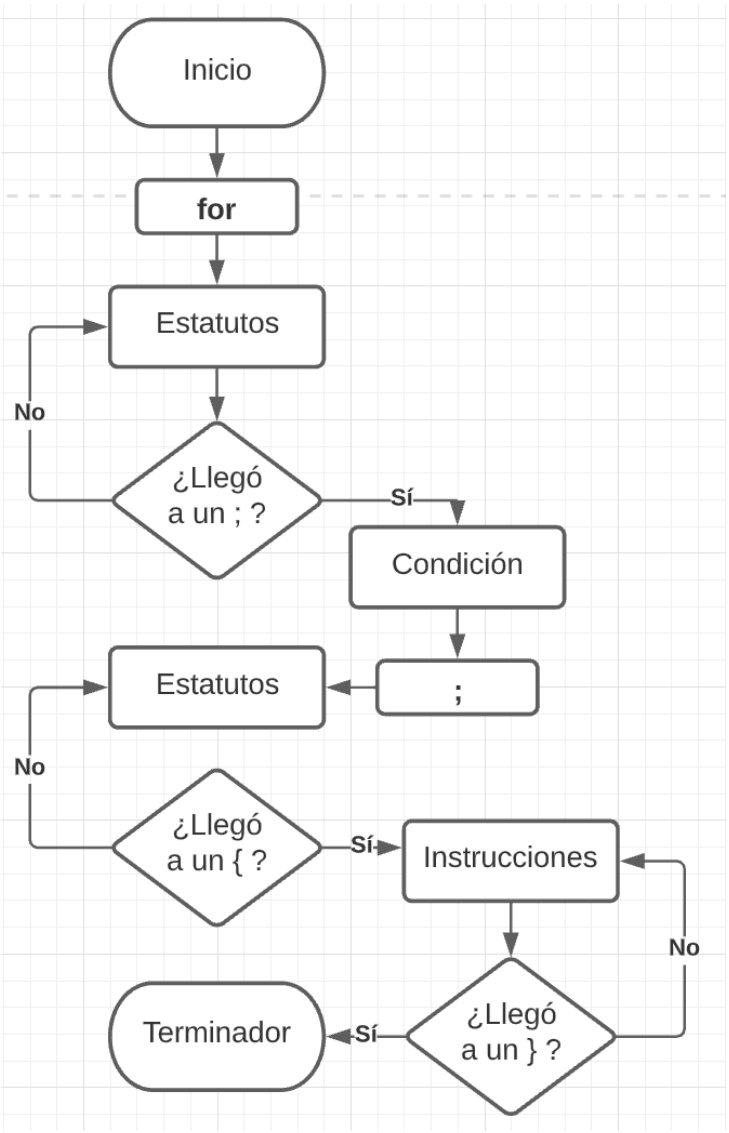


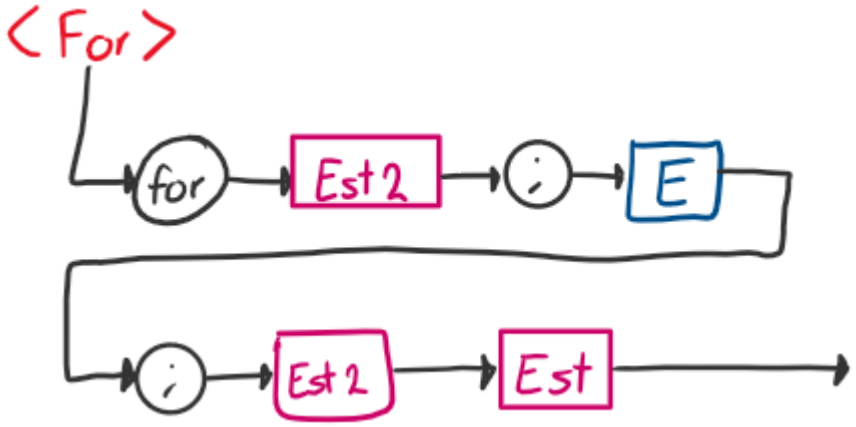
b. FOR

for i => 10; i < 20; i => i+1 **{**

~ Estatutos

}



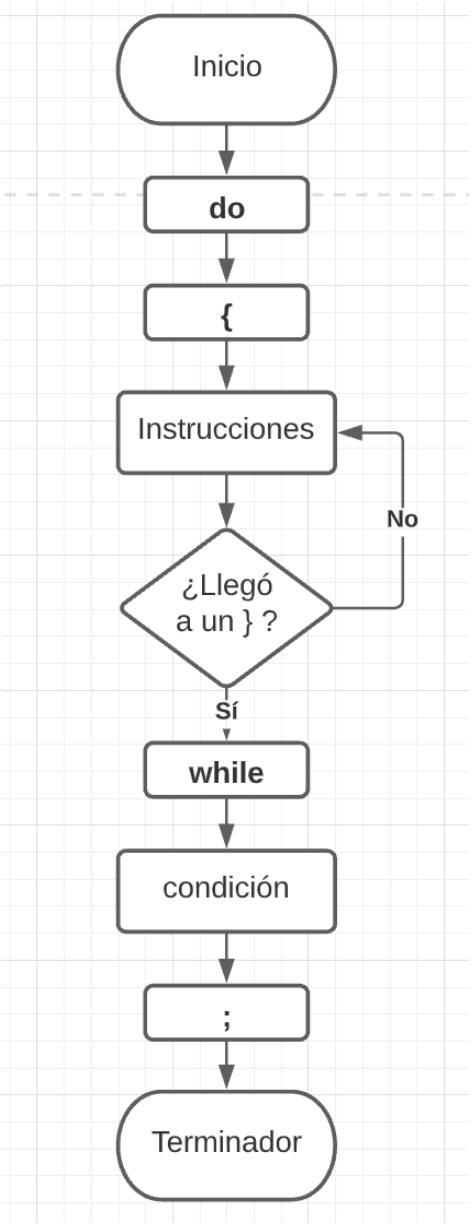


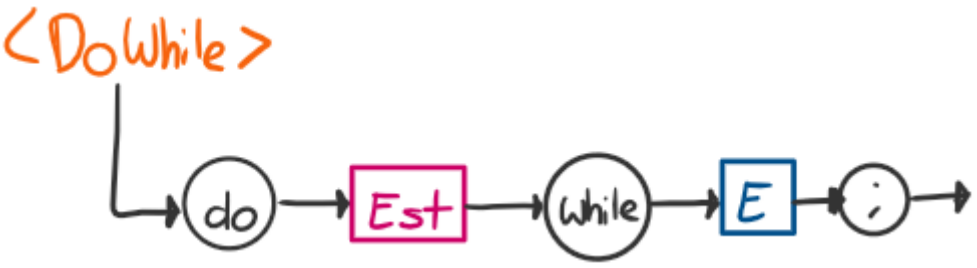
c. DO – WHILE

do {

~ Estatutos

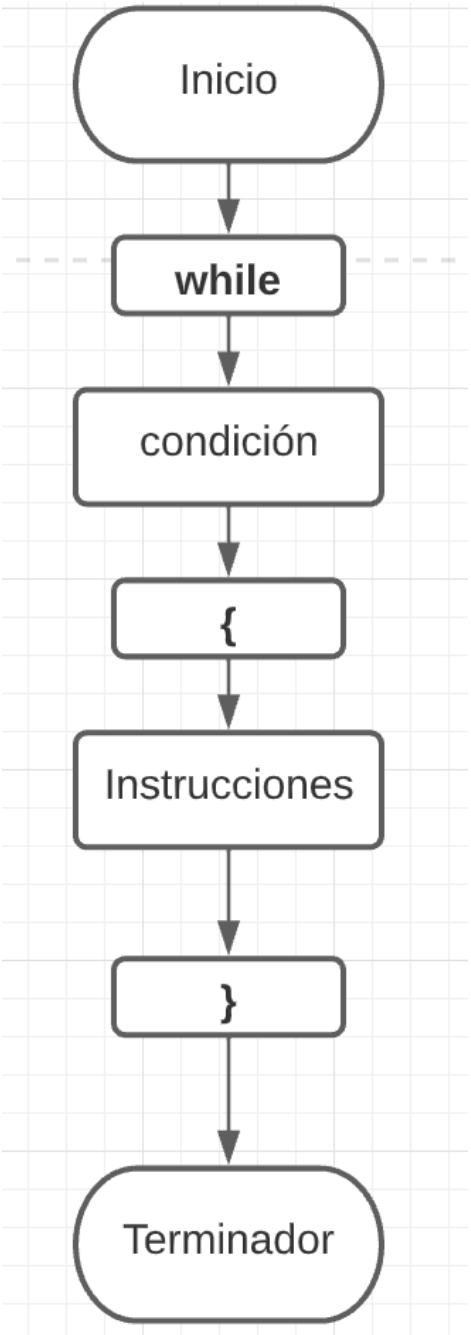
} **while** x < y;

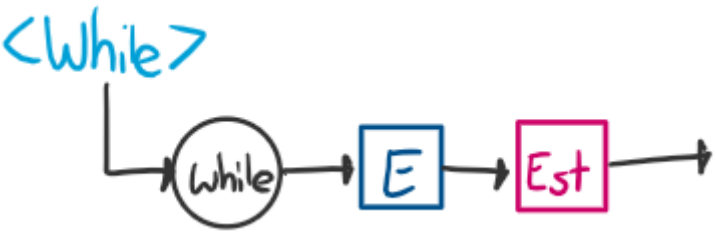




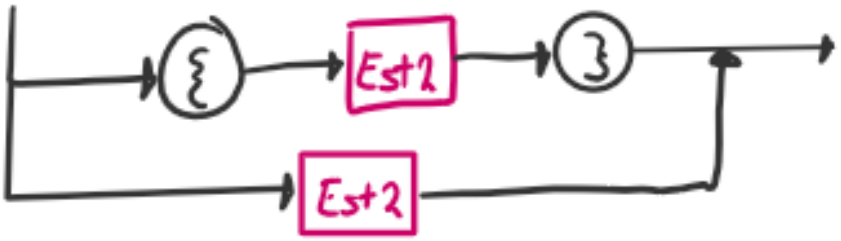
d. WHILE

```
while x < y {
        ~ Estatutos
}
```

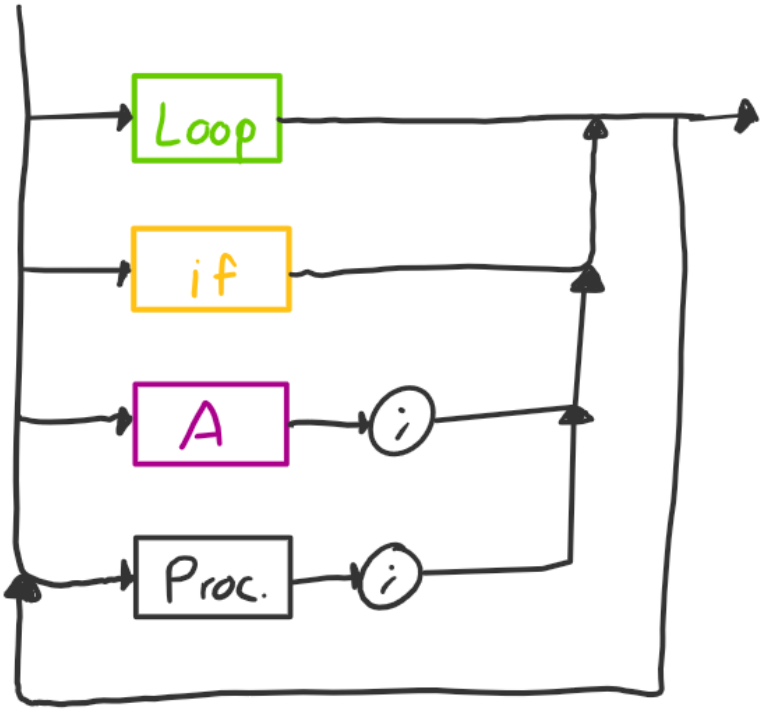




<Est>

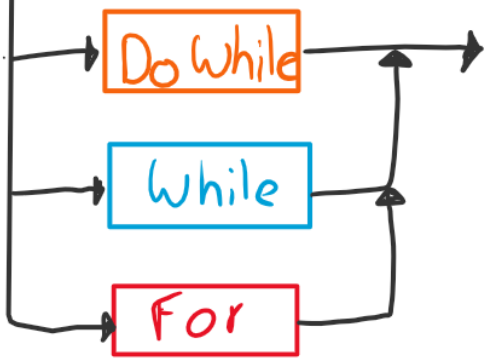


<Est2>



*Proc. es el nombre de un procedimiento

<Loop>



10. Lectura y escritura

    lectura → x => **input**(string);

    escritura → **print**(name_of_variable);

    Permitir concatenar strings y me gustaría poder convertir tipos como de int → string

    print("El valor es" + string(number));

11. Llamada entre funciones y procedimientos y valores de retorno para las funciones.

    Para el uso de funciones y procedimientos me gustaría adaptar la idea de las funciones en C++; es decir, clarificar el valor de retorno y mandar llamar las funciones, guardando en otra variable el retorno de la función en caso de que lo tenga.

    Ejemplo:

```
float myModule(float param1, float param2) {
        float : result;          ~ Definición de variable.
begin;
        result => param1 + param2;
        return result;           ~ Retorno de valor tipo float.
}

procedure main( ) {
        float : return_value;          ~ Definición de variable.
begin;
        return_value=> myModule(2.2, 4.4);  ~ Llamado a función.
        print(return_value);
        return;                        ~ Opcional.
}
```

Muestre un programa prueba hecho en su lenguaje (a mano o en computadora) que lea elemento por elemento, los números de dos matrices de máximo 5x5 (lea la cantidad de renglones y columnas de ambas matrices). Utilice funciones en su programa.

a. Verifique que las dos matrices puedan sumarse.
b. Calcule la suma de las dos matrices y deje la suma en una tercera matriz.
c. Imprima esta matriz.

```
float matrix1[5][5];
float matrix2[5][5];
float matrix3[5][5];

procedure enterMatrix(int number_of_rows, int number_of_columns, float matrix) {
        int row;
        int column;
begin;
        for row => 0; row < number_of_rows; row => row+1 {
                for column => 0; column < number_of_columns; column=> column+1 {
                        matrix[row][column] => input("Please enter the value of the element [" 
                        + string(row) + "][" + string(column) + "] of the matrix");
                }
        }
}

procedure addMatrix(int number_of_rows, int number_of_columns) {
        int row;
        int column;
begin;
        for row => 0; row < number_of_rows; row => row+1 {
                for column => 0; column < number_of_columns; column=> column+1 {
                        matrix3[row][column] => matrix1[row][column] + matrix2[row][column];
                }
        }
}

procedure printMatrix(int number_of_rows, int number_of_columns) {
        int row;
        int column;
begin;
        for row => 0; row < number_of_rows; row => row+1 {
                for column => 0; column < number_of_columns; column=> column+1 {
                        print(matrix[row][column]);
```

```
                print(" ");
            }
            print("\n");     ~ Salto de línea.
        }
}


procedure main( ) {
        int number_of_rows1;
        int number_of_rows2;
        int number_of_clumns1;
        int number_of_clumns2;
begin;
        number_of_rows1 => input("Enter the number of rows of the first robot");
        number_of_clumns1 => input("Enter the number of columns of the first robot");
        number_of_rows2 => input("Enter the number of rows of the second robot");
        number_of_clumns2 => input("Enter the number of columns of the second robot");

        ~ Verificar que ambas matrices puedan sumarse.
        if number_of_rows1 == number_of_rows2 & number_of_columns1 ==
        number_of_columns2 {
            enterMatrix(number_of_rows1, number_of_columns1, matrix1);
            enterMatrix(number_of_rows2, number_of_columns2, matrix2);
            ~ Sumar matrices y guardarlas en 3ra matriz.
            addMatrixes(number_of_rows1, number_of_columns1);
            ~ Imprimir matriz 3.
            print("Matrix3 = Matrix1 + Matrix2);
            print("\n\n");
            print("Matrix 1:\n");
            printMatrix(matrix1);
            print("Matrix 2:\n");
            printMatrix(matrix2);
            print("Matrix 3:\n");
            printMatrix(matrix3);
        } else {
            print("The matrix addition is not possible");
        }
}

main();
```