Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Organización de Lenguajes y Compiladores 1

Escuela de Vacaciones

Ing. Mario Bautista

Aux. Francisco Puac

# MANUAL DE USUARIO

## Proyecto

## JPR


Realizado por:

3149675670901 - Jose Alejandro Barrios Rodas

Fecha: 4 de Julio de 2021

## 1. DESCRIPCIÓN GENERAL

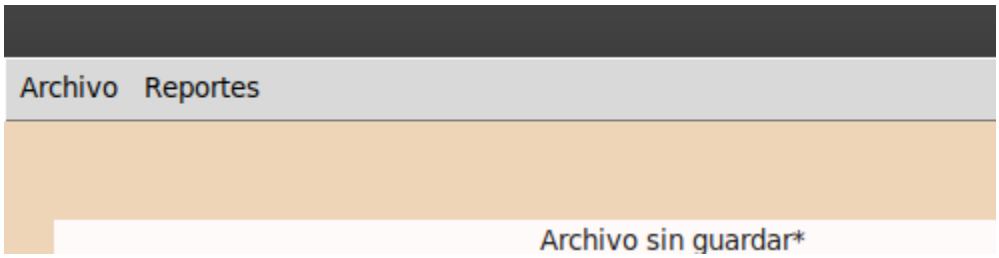
El editor de texto JPR hace uso de un interprete que ejecuta instrucciones de alto nivel definidas en el lenguaje JPR, este tiene como propósito que los alumnos acorde a la reforma curricular puedan ver las funcionalidades principales y la aplicación de los fundamentos de programación.


## 2. FUNCIONES BÁSICAS

- Crear archivos: El editor deberá ser capaz de crear archivos en blanco.
- Abrir archivos: El editor deberá abrir archivos .jpr
- Guardar: El editor deberá guardar el estado del archivo en el que se estará

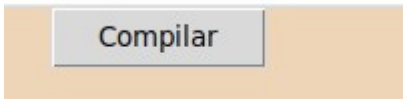
trabajando.

- Guardar Como: El editor deberá guardar el estado del archivo en el que se estará
  trabajando con un nuevo nombre a elegir.




## 3. BOTON DE ANALIZAR

Hará el llamado al intérprete, el cual se hará cargo de realizar los análisis léxico, sintáctico y semántico, además de ejecutar todas las sentencias.




## 4. BOTON DE REPORTES

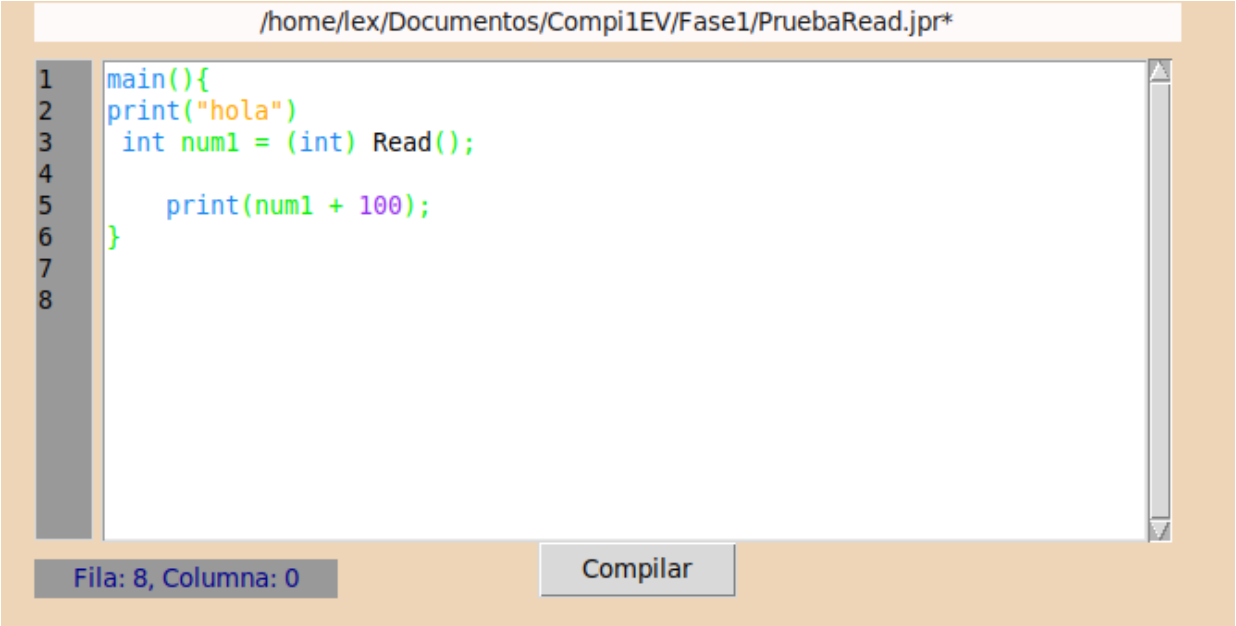
- Reporte de Errores: Se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico.
- Generar Árbol AST (Árbol de Análisis Sintáctico): se debe generar una imagen del árbol de análisis sintáctico que se genera al realizar los análisis.

- Reporte de Tabla de Símbolos: Se mostrarán todas las variables, métodos y funciones que han sido declarados dentro del flujo del programa.

## 5. CARACTERISTICAS

- Contador de Líneas: El editor deberá poder visualizar las líneas en donde se encuentra el código, para mejorar la búsqueda de errores.
- Posición del Cursor: El editor deberá poder visualizar el cursor en donde se encuentre activo en la entrada de código, para mejorar la búsqueda de errores.
- Se debe de pintar los tokens del código de entrada, el color de cada token se define en la siguiente tabla:

| Token | Color |
|---|---|
| Palabras reservadas | Azul |
| Cadenas, caracteres | Naranja |
| Números | Morado |
| Comentarios | Gris |
| Otro | Negro |




## 6. LENGUAJE

### Tipos de Datos Primitivos

Los tipos de dato que soportará el lenguaje en concepto de un tipo de variable se definen a continuación:

| TIPO | DEFINICION | DESCRIPCION | EJEMPLO | OBSERVACIONES |
|---|---|---|---|---|
| Entero | Int | Este tipo de datos aceptará solamente números enteros. | 1, 50, 100, 25552, etc. | Del -9.223.372.036.854.775.808 al 9.223.372.036.854.775.807 |
| Doble | Double | Admite valores numéricos con decimales. | 1.2, 50.23, 00.34, etc. | Se manejará cualquier cantidad de decimales |
| Booleano | Boolean | Admite valores que indican verdadero o falso. | true, false | Si se asigna un valor booleano a un entero se tomará como 1 o 0 respectivamente. |
| Caracter | Char | Tipo de dato que únicamente aceptará un único carácter, y estará delimitado por comillas simples. ' ' | 'a', 'b', 'c', 'E', 'Z', '1', '2', '^', '%', ')', '=', '!', '&', '/', '\\', '\n', etc. | En el caso de querer escribir comilla simple escribir se escribirá \ y después comilla simple \', si se quiere escribir \ se escribirá dos veces \\, existirá también \n, \t, \r, \", \'. |
| Cadena | String | Es un grupo o conjunto de caracteres que pueden tener cualquier carácter, y este se encontrará delimitado por comillas dobles. " " | "cadena1", "-- ** cadena 1" | Se permitirá cualquier carácter entre las comillas dobles, incluyendo los caracteres especiales con secuencias de escape:<br>\" comilla doble<br>\\ barra invertida<br>\n salto de línea<br>\r retorno de carro<br>\t tabulación |
| **Nulo** | Null | Este tipo de dato representa la ausencia de valor. | null | Cuando una variable solamente se declara, se le asignará automáticamente este valor. El valor nulo también se podrá asignar a una variable manualmente. |

## Caracteres Especiales

Las secuencias de escape se utilizan para definir ciertos caracteres especiales dentro de cadenas de texto. Las secuencias de escape disponibles son las siguientes:

| SECUENCIA | DESCRIPCION | EJEMPLO |
|---|---|---|
| \n | Salto de línea | "Hola\nMundo" -> "Hola Mundo" |
| \\ | Barra invertida | "C:\\miCarpeta\\Personal" -> "C:\miCarpeta\Personal" |
| \" | Comilla doble | \"Esto es una cadena\" -> "Esto es una cadena" |
| \t | Tabulación | "\tEsto es una tabulación" -> " Esto es una tabulación" |
| \' | Comilla Simple | \'Estas son comillas simples\' -> 'Estas son comillas simples' |

## Tipos de Datos complejos (Por referencia)

- Arreglos

## Operadores Aritméticos:

## Suma:

Es la operación aritmética que consiste en realizar la suma entre dos o más valores. El símbolo a utilizar es el signo más (+).

Observaciones:

- Al sumar dos datos numéricos (int o double), el resultado debes ser numérico.

Especificaciones de la operación suma

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

| + | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Int | Double | Int | | String |
| Double | Double | Double | Double | | String |
| Boolean | Int | Double | Int | | String |
| Char | | | | String | String |
| String | String | String | String | String | String |

**Nota:** Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.

**Resta**

Es la operación aritmética que consiste en realizar la resta entre dos o más valores. El símbolo por utilizar es el signo menos (-).

Observaciones:

- Al restar dos datos numéricos (int o double), el resultado debes ser numérico.

| - | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Int | Double | Int | | |
| Double | Double | Double | Double | | |
| Boolean | Int | Double | | | |
| Char | | | | | |
| String | | | | | |

**Multiplicación**

Operación aritmética que consiste en sumar un número (multiplicando) tantas veces como

indica otro número (multiplicador). El sino para representar la operación es el asterisco
(*).

Observaciones:

• Al multiplicar dos datos numéricos (entero o doble), el resultado debes ser numérico.

Especificaciones de la operación multiplicación

A continuación, se especifica en una tabla los resultados que se deberán obtener con
esta operación.

| * | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Int | Double | | | |
| Double | Double | Double | | | |
| Boolean | | | | | |
| Char | | | | | |
| String | | | | | |

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado
un error de tipo semántico.

**División**

Operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce
como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de
cociente. El operador de la división es la diagonal (/).

Observaciones:

• Al dividir dos datos numéricos (entero o doble), el resultado debes ser numérico.

Especificaciones de la operación división

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

| / | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Double | Double | | | |
| Double | Double | Double | | | |
| Boolean | | | | | |
| Char | | | | | |
| String | | | | | |

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico

**Potencia**

Es una operación aritmética de la forma a ** b donde a es el valor de la base y b es el valor del exponente que nos indicará cuantas veces queremos multiplicar el mismonúmero. Por ejemplo 5**3, a=5 y b=3 tendríamos que multiplicar 3 veces 5 para obtener el resultado final; 5x5x5 que da como resultado 125. Para realizar la operación se

utilizará el signo (**).

Observaciones:

• Al potenciar dos datos numéricos (entero o doble), el resultado debes ser numérico.

Especificaciones de la operación potencia

A continuación, se especifica en una tabla los resultados que se deberán obtener con

esta operación.

| ** | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Int | Double | | | |
| Double | Double | Double | | | |
| Boolean | | | | | |
| Char | | | | | |
| String | | | | | |

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado

un error de tipo semántico.

**Módulo**

Es una operación aritmética que obtiene el resto de la división de un numero entre otro.

El signo a utilizar es el porcentaje (%).

Observaciones:

• Al obtener el módulo de un numero el resultado debe ser numérico.

Especificaciones de la operación módulo

A continuación, se especifica en una tabla los resultados que se deberán obtener con

esta operación.

| % | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Double | Double | | | |
| Double | Double | Double | | | |
| Boolean | | | | | |
| Char | | | | | |
| String | | | | | |

**Negación Unaria**

Es una operación que niega el valor de un número, es decir que devuelve el contrario

del valor original, por ejemplo

Observaciones:

• La negación de un tipo entero o decimal debe generar como resultado su mismo

tipo.

Especificaciones de la operación negación

A continuación, se especifica en una tabla los resultados que se deberán obtener con

esta operación.

| -num | Resultado |
|---|---|
| Int | Int |
| Double | Double |
| Boolean | |
| Char | |
| String | |

## Operadores Relacionales

Son los símbolos que tienen como finalidad comparar expresiones, dando como resultado
valores booleanos. A continuación, se definen los símbolos que serán aceptados dentro del
lenguaje:

### Igualación

Compara ambos valores y verifica si son iguales (==).

| == | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Boolean | Boolean | | | Boolean |
| Double | Boolean | Boolean | | | Boolean |
| Boolean | | | Boolean | | Boolean |
| Char | | | | Boolean | |
| String | Boolean | Boolean | Boolean | | Boolean |

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado
un error de tipo semántico.

### Diferenciación

Compara ambos lados y verifica si son distintos (=!).

| =! | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Boolean | Boolean | | | Boolean |
| Double | Boolean | Boolean | | | Boolean |
| Boolean | | | Boolean | | Boolean |
| Char | | | | Boolean | |
| String | Boolean | Boolean | Boolean | | Boolean |

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado
un error de tipo semántico.

**Menor Que**

Compara ambos lados y verifica si el izquierdo es menor que el derecho (<).

| < | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Boolean | Boolean | | | |
| Double | Boolean | Boolean | | | |
| Boolean | | | Boolean | | |
| Char | | | | | |
| String | | | | | |

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado
un error de tipo semántico

**Mayor Que**

Compara ambos lados y verifica si el izquierdo es mayor que el derecho (>).

| > | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Boolean | Boolean | | | |
| Double | Boolean | Boolean | | | |
| Boolean | | | Boolean | | |
| Char | | | | | |
| String | | | | | |

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado
un error de tipo semántico.

**Menor Igual**

Compara ambos lados y verifica si el izquierdo es menor o igual que el derecho (<=)

| <= | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Boolean | Boolean | | | |
| Double | Boolean | Boolean | | | |
| Boolean | | | Boolean | | |
| Char | | | | | |
| String | | | | | |

**Mayor Igual**

Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho (>=).

| >= | Int | Double | Boolean | Char | String |
|---|---|---|---|---|---|
| Int | Boolean | Boolean | | | |
| Double | Boolean | Boolean | | | |
| Boolean | | | Boolean | | |
| Char | | | | | |
| String | | | | | |

**Nota:** Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.

## Operadores Lógicos

Son los símbolos que tienen como finalidad comparar expresiones a nivel lógico (verdadero o falso). A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

| OPERADOR | DESCRIPCIÓN | EJEMPLO | OBSERVACIONES |
|---|---|---|---|
| \|\| | **OR**: Compara expresiones lógicas y si al menos una es verdadera entonces devuelve verdadero en otro caso retorna falso | (false)\|\| bandera==true<br><br>Devuelve true | bandera es true |
| && | **AND**: Compara expresiones lógicas y si son ambas verdaderas entonces devuelve verdadero en otro caso retorna falso | flag1 && ("hola" == "hola")<br><br>Devuelve true | flag1 es true |
| ! | **NOT**: Devuelve el valor | !var1 | var1 es true |

| | | |
|---|---|---|
| | inverso de una expresión lógica si esta es verdadera entonces devolverá falso, de lo contrario retorna verdadero. | Devuelve falso |

### Signos de Agrupación

Los signos de agrupación serán utilizados para agrupar operaciones aritméticas, lógicas o relacionales. Los símbolos de agrupación están dados por paréntesis ( ).

### Precedencia de Operaciones

La precedencia de operadores nos indica la importancia en que una operación debe realizarse por encima del resto. A continuación, se define la misma.

| NIVEL | OPERADOR | ASOCIATIVIDAD |
|---|---|---|
| 0 | - | Derecha |
| 1 | ** | No asociativa |
| 2 | /, *, % | Izquierda |
| 3 | +, - | Izquierda |
| 4 | ==, =!, <, <=, >, >= | Izquierda |
| 5 | ! | Derecha |
| 6 | && | Izquierda |
| 7 | \| \| | Izquierda |

**NOTA:** el nivel 0 es el nivel de mayor importancia.

## Caracteres de finalización y encapsulamiento de sentencias

El lenguaje se verá restringido por dos reglas que ayudan a finalizar una instrucción y encapsular sentencias:

- **Finalización de instrucciones:** para finalizar una instrucción **puede o no venir** el signo punto y coma **(;).**
- **Encapsular sentencias:** para encapsular sentencias dadas por los ciclos, métodos, funciones, etc, se utilizará los signos **{** y **}**.

```
                    #Ejemplo de
finalización de instrucción
var edad = 18
                #Ejemplo de encapsulamiento de sentencias


If(i==1){
var a=15;
                        print("soy el numero "+a)
                }
```

## Declaración y asignación de variables

Una variable deberá de ser declarada antes de poder ser utilizada. Todas las variables tendrán un tipo de dato y un nombre de identificador. Las variables podrán ser declaradas global y localmente. La declaración de variables debe de tener la palabra "var", seguido de un identificador y su

expresión. Cuando se quiera declarar o asignar una variable que tenga un arreglo, esta será tomada por **referencia**.

```
                    'var' identificador
                    | 'var' identificador '=' <EXPRESION>

#Ejemplos
var numero;   # null
                    var
cadena = "hola"     #String
var var_1 = 'a';          # char
var verdadero;         # null
```

Las variables solamente declaradas tendrán el valor null por defecto, al asignarse otro valor, las variables no podrán cambiar de tipo de dato, **solamente si se asigna nuevamente un valor null**.

```
                    identificador = <EXPRESION>
                    #Ejemplos

                    Var
numero;              #null
numero = 4;                #int
var verdadero;           #null
verdadero = true       #boolean
var valor = numero; # int 4
valor = null                  #null
```

```
                                valor = “cadena”        #string cadena
                                var cadena = “valor = “ + valor;
#cadena = “valor = cadena”
```

## Casteos

Los casteos son una forma de indicar al lenguaje que convierta un tipo de dato en otro, por lo que, si queremos cambiar un valor a otro tipo, es la forma adecuada de hacerlo. Para hacer esto, se colocará la palabra reservada del tipo de dato destino entre paréntesis seguido de una expresión.

‘(’ <TIPO> ‘)’ <EXPRESION>

El lenguaje aceptará los siguientes casteos:

- **Int a double**
- **Double a Int**
- **Int a String**
- **Int a Char**
- **Double a String**
- **Char a int**
- **Char a double**
- **String a Int**
- **String double**
- **String a boolean**

```
'('<TIPO>')' <EXPRESION>

#Ejemplos

var edad = (int) 18.6;        #toma el valor
entero de 18

var letra = (char) 70          #tomar el
valor 'F' ya que el 70 en ascii es F                    var
numero = (double) 16; #toma el valor 16.0
```

## Incremento y Decremento

Los incrementos y decrementos nos ayudan a realizar la suma o resta continua de una variable de uno en uno, es decir si incrementamos una variable, se incrementará de uno en uno, mientras que, si realizamos un decremento, hará la operación contraria. Solamente realizará operaciones con tipo de dato Int y Double.

```
<IDENTIFICADOR>'+"+'
<IDENTIFICADOR> '-"-'

#Ejemplos
var edad = 18.5;
edad++; #tiene el valor de 19.5
edad--; #tiene el valor 18.5


var anio=2020;
```

```
anio = 1 + anio++; #aumenta una unidad a la variable año y luego obtiene el valor de 2022
anio = anio--;    #obtiene el valor de 2021
```

## Arreglos

Los arreglos son una estructura de datos de tamaño fijo que pueden almacenar valores de forma limitada, y los valores que pueden almacenar son de un único tipo; int, double, boolean, char o string. **El lenguaje permite el uso de arreglos de múltiples dimensiones.**

**Observaciones:**

- La posición de cada vector será N-1. Por ejemplo, si deseo acceder al primer valor de un arreglo debo acceder como val[0].

## Declaración de Arreglos

Al momento de declarar un arreglo, tenemos dos tipos que son:

- **Declaración tipo 1:** En esta declaración, se indica por medio de una expresión numérica del tamaño y la dimensión que se desea el arreglo, tomando el valor null para cada espacio. Sin embargo, estos valores no podrán cambiar de tipo.
- **Declaración tipo 2:** En esta declaración, se indica por medio de una lista de valores separados por coma, los valores que tendrá el arreglo, en este caso el tamaño del arreglo será el de la misma cantidad de valores de la lista. Si es de más dimensiones el arreglo, la lista deberá de contener las

listas necesarias para la asignación de todos los elementos del arreglo.

#DECLARACION TIPO 1

<TIPO> ( '[' ']' )+ <ID> = new <TIPO> ( '[' <EXPRESION> ']' )+

#DECLARACION TIPO 2

<TIPO> ( '[' ']' )+ <ID> = '{' <LISTAVALORES> '}'

#Ejemplo de declaración tipo 1

int[ ] arr1 = new int[4]; #se crea un arreglo de 4 posiciones, con null en cada posición int[ ][ ] matriz1 = new int[2][2]; #se crea un arreglo de 4 posiciones, con null en cada posición

int[ ][ ] matriz2 = matriz1; # la referencia de matriz1 está en matriz2

#Ejemplo de declaración tipo 2

string[ ] arreglo2 = {"hola", "Mundo"}; #arreglo de 2 posiciones, con "Hola" y "Mundo" int[ ][ ] matriz = { {11, 12} , {21,22} }; #arreglo de 2 dimensiones, con 2 posiciones en cada dimensión.

char[ ][ ][ ] cubo = { { {'a', 'b'} , {'c','d'} } , { {'e', 'f'} , {'g','h'} } } , { { {'i', 'j'} , {'k', 'l'} } , { {

} #arreglo de 3 dimensiones de 2 dimensiones, con 2 posiciones en cada dimensión. ([3][2][2])

char[ ][ ][ ] cubo2 = { { {'a', 'b'} , {'c','d'} , {'c','d'} } , { {'e', 'f'} , {'g','h'} , {'g','h'} } , { {'i', 'j'} , {'k','l'} , {'k','l'} } , { {'a', 'b'} , {'c','d'} , {'c','d'} } } #arreglo de 3 dimensiones de 2 dimensiones, con 2 posiciones en cada dimensión. ([4][3][2])

## Acceso a Arreglos

Para acceder al valor de una posición de un arreglo, se colocará el nombre del arreglo seguido de '[' EXPRESION ']'.

<ID> ( '[' EXPRESION ']' )+

#Ejemplo de acceso

string[ ] arr2 = {"hola", "Mundo"}; #creamos un arreglo de 2 posiciones de tipo string string { {'a', 'b'} , {'c','d'} } , { {'e', 'f'} , {'g','h'} } , { {'i', 'j'} , {'k', 'l'} }

valorPosicion = arr2 [0] #posición 0, valorPosicion = "hola"

char[ ][ ][ ] cubo = { { } }; char caracter = cubo[2][0][1];

# caracter = 'j'

## Modificación de Arreglos

Para modificar el valor de una posición de un arreglo, se debe colocar el nombre del arreglo seguido de '[' EXPRESION ']' = EXPRESION

<ID> ( '[' EXPRESION ']' )+ = EXPRESION

```
string[ ] arr2 = {"hola", "Mundo"}; #arreglo de 2 posiciones, con "Hola" y "Mundo" int[ ] arrNumero = {2020,2021,2022}; arr2 [0] = "OLC1";
arr2 [1] = "1er Semestre "+ arrNumero [1];

#*

RESULTADO arr2
[0]= "OLC1 "
arr2 [1]= "1er Semestre 2021"
*#
char[ ][ ][ ] cubo = { { {'a', 'b'} , {'c','d'} } , { {'e', 'f'} , {'g','h'} } , { {'i', 'j'} , {'k','l'} } };
cubo[2][0][1] = 'z'; # cubo = { { {'a', 'b'} , {'c','d'} } , { {'e', 'f'} , {'g','h'} } , { {'i', 'z'} , {'k','l'} } };
```

```
int[ ] matrizNumero = { {1,2}, { 3, 4 } };


cubo[ matrizNumero[0][0] ][ matrizNumero[0][1] -1 ][ matrizNumero[0]
[0] ] = 'z'
                                #1          #1
#1 #cubo = { { {'a', 'b'} , {'c','d'} } , 'z' { {'e', 'f'} , {'g',} }
, { {'i', 'z'} , {'k','l'} } };
```

4.17. Sentencias de control

Estas sentencias modifican el flujo del programa introduciendo condicionales. Las sentencias de control para el programa son el IF y el SWITCH.

**OBSERVACIONES:**

- También, entre las sentencias pueden tener Ifs anidados.

4.17.1. if

La sentencia if ejecuta las instrucciones sólo si se cumple una condición. Si la condición es falsa, se omiten las sentencias dentro del if.

4.17.1.1. if

```
                    'if' '(' [<EXPRESION>] ')' '{'
            [<INSTRUCCIONES>]
        '}'
                    | 'if' '(' [<EXPRESION>] ')' '{'
            [<INSTRUCCIONES>]
```

```
                                        '}' 'else' '{'
                        [<INSTRUCCIONES>]
                    '}'
                                        | 'if' '(' [<EXPRESION>]   ')' '{'
                        [<INSTRUCCIONES>]

                '}' 'else'
                [<IF>]

#Ejemplo de cómo se implementar un ciclo if

if (x <50)
 {
       Print("Menor que 50");
       #Más sentencias
}
```

4.17.1.2. else

```
//Ejemplo de cómo se implementar un ciclo if-else

if (x <50)
 {
       print("Menor que 50");
       //Más sentencias
}
else
{
```

```
        print("Mayor que 100");
        //Más sentencias
}
```

4.17.1.3. else if

```
#Ejemplo de cómo se implementar un ciclo else if

if (x > 50)
  {
        print("Mayor que 50");
        #Más sentencias
}
else if (x <= 50 && x > 0)
{
        print ("Menor que 50");
        if (x > 25)
        {
                print("Número mayor que 25");
                #Más sentencias
        }
else
        {
                print("Número menor que 25");
                #Más sentencias
        }
```

```
        #Más sentencias
}
else
{
        print("Número negativo");
        #Más sentencias
}
```

### 4.17.2. Switch Case

Switch case es una estructura utilizada para agilizar la toma de decisiones múltiples, trabaja de la misma manera que lo harían sucesivos else-if.

#### 4.17.2.1. Switch

Estructura principal del switch, donde se indica la expresión a evaluar.

```
                    'switch' '(' [<EXPRESION>] ')' '{'
        [<CASES_LIST>] [<DEFAULT>]
    '}'
                    | 'switch' '('<EXPRESION> ')' '{'
        [<CASES_LIST>]
    '}'
                    | 'switch' '('<EXPRESION> ')' '{'
        [<DEFAULT>]
    '}'
```

#### 4.17.2.2. Case

Estructura que contiene las diversas opciones a evaluar con la expresión establecida en el switch.

| |
|---|
| 'case' [<EXPRESION>] ':' |
| [<INSTRUCCIONES>] |

#### 4.17.2.3. Default

Estructura que contiene las sentencias si en dado caso no haya salido del switch por medio de una sentencia **break**.

| |
|---|
| 'default' ':' |
| [<INSTRUCCIONES>] |

```
# EJEMPLO DE SWITCH
var edad = 18;
switch( edad ) {
Case 10:
                    Print("Tengo 10 anios.");
                    # mas sentencias

Break;          Case
18:

Print("Tengo 18 anios.");
# mas sentencias
Case "25":
                    Print("Tengo 25 anios en String.");
```

```
                    #
mas sentencias
Break;          Default:
                    Print("No se que edad tengo. :(");
                    # mas sentencias
                    Break;
 }
#* Salida esperada
Tengo 18 anios.
No se que edad
tengo. :( *#
```

**OBSERVACIONES:**

- Si la cláusula "case" no posee ninguna sentencia "break", al terminar todas las sentencias del case ingresado, el lenguaje seguirá evaluando las demás opciones.

4.18. Sentencias cíclicas

Los ciclos o bucles, son una secuencia de instrucciones de código que se ejecutan una vez tras otra mientras la condición, que se ha asignado para que pueda ejecutarse, sea verdadera. En el lenguaje actual, se podrán realizar 2 sentencias cíclicas que se describen a continuación.

**OBSERVACIONES:**

- Es importante destacar que pueden tener ciclos anidados entre las sentencias a ejecutar.
- También, entre las sentencias pueden tener ciclos diferentes anidados.

### 4.18.1. While

El ciclo o bucle While, es una sentencia que ejecuta una secuencia de instrucciones mientras la condición de ejecución se mantenga verdadera.

```
                                    'while'
'('[<EXPRESION>] ')' '{'
                    [<INSTRUCCIONES>]
                '}'


        #Ejemplo de cómo se implementar un ciclo
while


   while (x<100){      if (x
> 50)
                    {
                        print("Mayor que 50");

#Más sentencias
                    }
```

```
                    else
                    {
                         print("Menor que 50");

#Más sentencias
                    }
                                                X++;

#Más sentencias
                                          }
```

4.18.2. For

El ciclo o bucle for, es una sentencia que nos permite ejecutar N cantidad de veces la secuencia de instrucciones que se encuentra dentro de ella.

**OBSERVACIONES:**

- Para la actualización de la variable del ciclo for, se puede utilizar:
  - **Incremento | Decremento:** i++ | i--
  - **Asignación:** como i=i+1, i=i-1, i=5, i=x, etc, es decir cualquier tipo de asignación.
- Dentro pueden venir N instrucciones

```
'for' '(' ([<DECLARACION>||<ASIGNACION>])';' [<CONDICION>]';'
```

```
[<[ACTUALIZACION>] ‘)’ ‘{‘ [<INSTRUCCIONES>]
‘}’

#Ejemplo 1: declaración
dentro del for con incremento for ( var i=0;
i<3;i++ ){ print(“i=”+i)

#más sentencias
}
```

```
#*RESULTADO

i=0
i=1
i=2
*#

#Ejemplo 2: asignación de variable previamente declarada y
decremento por asignación for ( i=5; i>2;i=i-1 ){
print(“i=”+i)
#más sentencias
}
#*RESULTADO

i=5
i=4
i=3
```

```
*#
```

## 4.19. Sentencias de transferencia

Las sentencias de transferencia nos permiten manipular el comportamiento de los bucles, ya sea para detenerlo o para saltarse algunas iteraciones. El lenguaje soporta las siguientes sentencias:

### 4.17.1. Break

La sentencia break hace que se salga del ciclo inmediatamente, es decir que el código que se encuentre después del break en la misma iteración no se ejecutara y este se saldrá del ciclo.

```
#Ejemplo en un ciclo for
for(var i = 0; i < 9; i+
+){
        if(i==5){
                print("Me salgo del ciclo en el
numero " + i);                break;
        }
        print(i);
}
```

### 4.17.2. Continue

La sentencia continue puede detener la ejecución de la iteración actual y saltar a la siguiente. La sentencia continue siempre debe de estar dentro de un ciclo, de lo contrario será un error.

```
#Ejemplo de continue en un ciclo for
for(var i = 0; i < 9; i++){
        if(i==5){
                print("Me salte el numero
" + i);                continue;
        }
        print(i);
        #mas sentencias
}
```

### 4.17.3. Return

La sentencia return finaliza la ejecución de un método o función y puede especificar un valor para ser devuelto a quien llama a la función. Puede ser devuelto cualquier dato primitivo, null o un arreglo.

```
                return <EXPRESION>


#Ejemplos
func mi_metodo(){
                        var i;
                        for(i = 0; i <
9; i++){
                                if(i==5){

return i; #se detiene y retorna 5
                                }
                                print(i);
                        }
                }
            func sumar(int n1, int
n2){

var n3;
n3 = n1+n2;
                    return n3;
#retorno el valor
            }
```

## 4.20. Funciones

Una función es una subrutina de código que se identifica con un nombre, un conjunto de parámetros y de instrucciones. Para este lenguaje las funciones serán declaradas indicando que serán funciones, luego un identificador para la función, seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no tenga parámetros).

Cada parámetro debe estar compuesto por su tipo seguido de un identificador, para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir la función y sus parámetros se declara el cuerpo de la función, el cual es un conjunto de instrucciones delimitadas por llaves {}.

Como observación, se podrán ingresar arreglos como parámetros en las funciones, donde se indica su tipo seguido de la nomenclatura de un arreglo vacío (arreglo[]) dependiendo en el número de dimensiones que tenga el arreglo.
La función puede llevar o no un valor de retorno, si no posee alguna sentencia return y no devuelve ningún valor, será considerado como un método.

‘func’ <ID> ‘(‘ [<PARAMETROS>] ‘)’ ‘{‘
[<INSTRUCCIONES>]

‘}’

PARAMETROS -> [<PARAMETROS] ‘,’ [<TIPO>] ([])* [<ID>]

| [<TIPO>] ([])* [<ID>]

#Ejemplo de declaración de una función de enteros

Func conversion(double pies, string tipo){ if (tipo == “metro”)

{

return pies/3.281;

}

else

{

return -1;

}

}

#Ejemplo de declaración de un método

func holamundo(){

print(“Hola mundo”);

}

Cabe a destacar que **no habrá sobrecarga de funciones y métodos** dentro de este lenguaje por lo que solo puede existir una función o método con el id declarado por lo que si se crea otra función o método con un id previamente utilizado esto debe de generar un error de tipo semántico.

## 4.21. Llamadas

Una llamada a una función cumple con la tarea de ejecutar el código de una función del código de entrada ingresado.

La llamada a una función puede devolver un resultado que ha de ser recogido, bien asignándolo a una variable del tipo adecuado o bien integrándolo en una expresión.

La sintaxis de las llamadas de las funciones es la siguiente:

```
LLAMADA -> [<ID>] '(' [<PARAMETROS_LLAMADA>] ')'
                  | [<ID>] '('  ')'

PARAMETROS_LLAMADA -> [<PARAMETROS_LLAMADA>] ','
[<EXPRESION>]
                                                  |
[<EXPRESION>]
        #Ejemplo de llamada de un método
```

```
                  Print("Ejemplo
de llamada a método");
```

```
holamundo(); #* Salida
esperada
    Ejemplo de llamada a método
    Hola Mundo
*#
#Ejemplo de llamada de una función

                    Print("Ejemplo de
llamada a función"); var num =
suma(6,5)    #num = 11 Print("El
valor de num es: " + num);
#* Salida esperada
    Ejemplo de llamada a función
    Aquí puede venir cualquier sentencia :D
    El valor de num es: 11
*#

func suma(int num1, int num2)
{
        print("Aquí puede venir cualquier
sentencia :D");          return num1 + num2;
        print("Aquí pueden venir más sentencias, pero no se
ejecutarán por la
sentencia RETURN D:"); #print en una línea
}
```

**OBSERVACIONES:**

- Al momento de ejecutar cualquier llamada, no se diferenciarán entre métodos y funciones, por lo tanto, podrá venir una función que retorne un valor como un método, pero la expresión retornada no se asignará a ninguna variable.
- Se podrán llamar métodos y funciones antes que se encuentren declaradas, para ello se recomienda realizar 2 pasadas del AST generado: La primera para almacenar todas las funciones, y la segunda para las variables globales y la función main.
- Se pueden enviar arreglos como parámetros en la llamada.

**Función Print**

Esta función nos permite imprimir expresiones con valores únicamente de tipo int, double, booleano, string y char.

```
'print' '(' <EXPRESION> ')'
#Ejemplo
print("Hola mundo!!")
print("Sale compi \n" +
valor);                    print(suma(2,2))
```

**Función Read**

Esta función nos permite obtener valores que queramos ingresar en el momento de ejecución del código, **el valor ingresado se tomará como un string**, por lo que, si se quiere ingresar un número, para tomarlo como tal se debe de castearlo.

Se recomienda utilizar una pausa, poder escribir el valor requerido en el área de consola, y al momento de presionar "ENTER", que el programa capture el valor de la última línea de la consola y seguir con su ejecución.

```
'read' '(")'
#Ejemplo
Var input = read()
#Ingresa 100 (String)
Var numInput = (int)input                    # int 100
print("el valor de input es: " + input);   # el valor de input es: 100
```

Función toLower

Esta función recibe como parámetro una expresión de tipo **String** y retorna una nueva cadena con todas las letras minúsculas.

```
'toLower' '(' <EXPRESION> ')'
#Ejemplo
```

```
        var cad_1 = toLower("hOla MunDo");  # cad_1 = "hola
mundo"
        var cad_2 = toLower("RESULTADO = " + 100);  # cad_2 =
"resultado = 100"
```

4.25. Función toUpper

Esta función recibe como parámetro una expresión de tipo cadena retorna una nueva cadena con todas las letras mayúsculas.

```
        'toUpper' '(' <EXPRESION> ')'
        #Ejemplo
        var cad_1 = toUpper("hOla MunDo");  # cad_1 = "HOLA
MUNDO"
        var cad_2 = toUpper("resultado = " + 100)  # cad_2 =
"RESULTADO = 100"
```

**Funciones nativas**

**Length**

Esta función recibe como parámetro un arreglo o una cadena y devuelve el tamaño de este.

```
'length' '(' <EXPRESION> ')'

# En donde < EXPRESION > puede ser:
#    -arreglo linealizado
#    -cadena
```

```
#Ejemplo
Int[ ] arr1 =new int[ 20 ];
string[ ] arr2 = {"hola",
"Mundo"};
var tam_arr1 = length(arr1); # tam_arr1 = 20

var tam_arr2 = length(arr2); // tam_arr2 = 2

var tam_hola = length(arr2[0]); // tam_hola = 4
```

**Nota**: Si recibe como parámetro un tipo de dato no especificado, se considera un error semántico.

**Truncate**

Esta función recibe como parámetro un valor numérico. Permite eliminar los decimales de un número, retornando un entero.

```
'truncate' '('<VALOR>')'

# Ejemplo

var nuevoValor = truncate(3.53); #
nuevoValor = 3 var otroValor =
```

```
truncate(10); # otroValor = 10

var decimal = 15.4654849;
var entero = truncate(decimal); # entero = 15
```

Nota: Si la función recibe un valor no numérico, se considera un error semántico.

**Round**

Esta función recibe como parámetro un valor numérico. Permite redondear los números decimales según las siguientes reglas: Si el decimal es mayor o igual que 0.5, se aproxima al número superior Si el decimal es menor que 0.5, se aproxima al número inferior. Retorna un valor entero.

```
'round' '(' <VALOR>')'

# Ejemplo

var valor = round(5.8);
//valor = 6 var valor2 =
round(5.4); //valor2 = 5
```

### Typeof

Esta función retorna una cadena con el nombre del tipo de dato evaluado.

| |
|---|
| 'typeof' '(' <VALOR> ')'<br><br>#Ejemplo |
| Int[][] arr2 =new int[3][2]<br>String tipo = typeof(15); # tipo = "INT"<br>String tipo2 = typeof(15.25) # tipo = "DOUBLE"<br>String tipo3 = typeof(arr2); # tipo3 = "ARREGLO->INT" |

### Main

Para poder ejecutar todo el código generado dentro del lenguaje, se utilizará la sentencia MAIN para indicar en dónde se iniciará a ejecutar el programa.

**Observaciones:**

- Puede venir solo una vez, si viene más de una vez deberá lanzar error y no podrá ejecutar ninguna instrucción.

```
                    'main' '(' ')' '{' <Instrucciones> '}'
                    #Ejemplo
                    # declaraciones y asignaciones de variables
globales
                    #
declaraciones de funciones
main(){
                        print("hola soy un mensaje");
                        #*
                            Muchas más
instrucciones, llamadas a funciones, etc
                        *#
                    }
                    # declaraciones de funciones
```

## Reportes

Los reportes son una parte fundamental de JPR, ya que muestra de forma visual las herramientas utilizadas para realizar la ejecución del código.

A continuación, se muestran ejemplos de estos reportes. (Queda a discreción del estudiante el diseño de estos, solo se pide que sean totalmente legibles).

### Tabla de Símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución. Se deberán mostrar todas las variables, funciones y métodos declarados, así como su tipo y toda la información que se considere necesaria.

| Identificador | Tipo | Tipo | Entorno | Valor | Línea | Columna |
|---|---|---|---|---|---|---|
| **Factor1** | Variable | Entero | Función multiplicar | null | 15 | 4 |
| **Factor2** | Variable | Decimal | Función multiplicar | 0.7 | 16 | 7 |
| **Resultado** | Variable | Decimal | Función multiplicar | 34.7 | 17 | 7 |
| **MostrarMensaje** | Función | -- | - | - | 50 | 6 |
| **Multiplicar** | Función | Decimal | - | - | 14 | 10 |

### Tabla de Errores

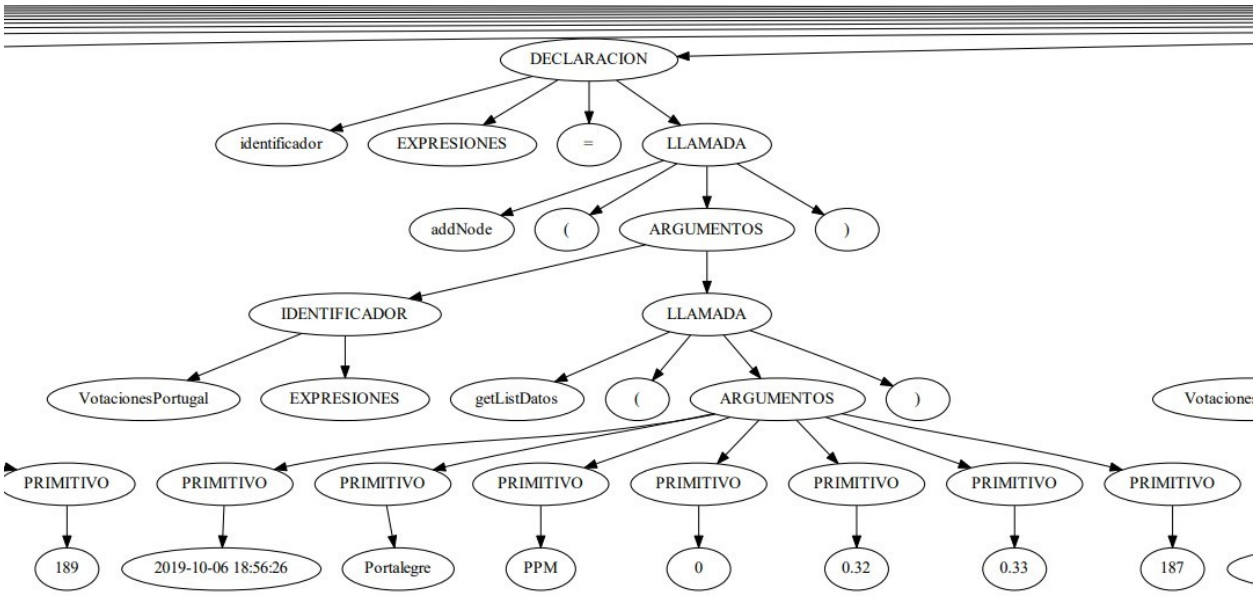
El reporte de errores debe contener la información suficiente para detectar y corregir errores en el código fuente.

| # | Tipo de Error | Descripción | Línea | Columna |
|---|---|---|---|---|
| **1** | Léxico | El carácter "$" no pertenece al lenguaje. | 7 | 32 |
| **2** | Sintáctico | Encontrado Identificador | 150 | 12 |

| | | | | |
|---|---|---|---|---|
| | | “Ejemplo”, se esperaba Palabra Reservada “Valor” | | |
| **3** | Semántico | No es posible operar String / Char | 200 | 23 |

## AST

Este reporte muestra el árbol de sintaxis producido al analizar los archivos de entrada. Este debe de representarse como un grafo. Se deben mostrar los nodos que el estudiante considere necesarios para describir el flujo realizado para analizar e interpretar sus archivos de entrada.



Nota: Se sugiere a los estudiantes utilizar la herramienta Graphviz para graficar su AST.

### Salidas en Consola

La consola es el área de salida del intérprete. Por medio de esta herramienta se podrán visualizar las salidas generadas por la función nativa “print”, así como los errores léxicos, sintácticos y semánticos.

```
> Este es un mensaje desde mi interprete de compi 1.
>
>
---> Error léxico: Símbolo "#" no reconocido en línea 10 y columna 7
---> Error Semántico: Se ha intentado asignar un entero a una variable booleana
```