Universidad Rafael Landívar
Facultad de Ingeniería
Compiladores
2019


# Manual Técnico y de Usuario Compilador Loop

**Marcos Josué Sierra Pac** **Carné: 15531-17**
**Sharon Rubí de los Ángeles Gómez Fuentes** **Carné: 15052-17**

# Tabla de contenido

# Introducción

La programación orientada a objetos es un paradigma de programación que vino a revolucionar la forma en la que se veía la programación y la forma en la que se obtenían resultados, esto se ha hecho patente en la gran versatilidad que ofrecen lenguajes orientados a objetos como Java o C++, siendo estas solo dos ejemplos de la gran cantidad de lenguajes disponibles para la programación enfocada a computadores o teléfonos móviles, sin embargo, existen pocas opciones de lenguajes de alto nivel orientados a objetos para la programación de pic, siendo lo más utilizado Mikroc o bien assembler propiamente, siendo un método bastante difícil de comprender para aquellos que tienen pocas bases de electrónica y programación. El lenguaje que se plantea en este proyecto, que lleva por nombre Loop, tiene por objetivo ser una opción simple y versátil para la programación orientada a objetos centrada en el pic 16f84A, facilitando a través de esta, la programación de estos microcontroladores.

Existe una gran varianza en el tamaño de los archivos escritos directamente en assembler y otros generados por un compiladores, esto debido a que un compilador debe prever de la forma mas general posible todos los casos y comandos que reciba, esto queda patente durante la guía técnica ya que se ve como existe una gran cantidad de comandos disponibles para las funciones, mientras tanto en assembler la cantidad de instrucciones es mucho más reducida y básica, ofreciendo las herramientas para hacer cualquier cosa pero de forma mas laboriosa, pues todos los procesos deben ser programados a mano.

A continuación, se describe la estructura y funcionamiento de la aplicación en dos secciones bien diferenciadas, en la primera sección del manual se describen los lineamientos que se tomaron en cuenta para la creación del compilador, el lenguaje usado para la aplicación, así como otros detalles técnicos de relevancia para comprender como se estructuro el programa y como funciona tanto el analizador léxico como el sintáctico. En la segunda sección principal se describe el funcionamiento de la aplicación, así como detalles de relevancia para el usuario como pueden ser las palabras reservadas del lenguaje, la estructura de este y el funcionamiento de la interfaz gráfica del compilador. Esto con el fin de poder ofrecer un conocimiento más profundo para el usuario que desee comprender mejor el funcionamiento del compilador y el lenguaje, pero tomando en cuenta también al usuario que desea simplemente usar el lenguaje de programación o la aplicación para compilar un archivo escrito en Loop.

# Manual Técnico

Esta sección tiene por objetivo analizar a fondo las características de la aplicación, así como las reglas que se han implementado para el análisis del lenguaje y la generación del código de salida, se divide en dos secciones principales las cuales son las características de la aplicación, donde se especifica la versión del lenguaje utilizada, así como el sistema operativo sobre el que se realizó el desarrollo y las pruebas de la aplicación, entre otras cosas. Luego encontramos la sección de detalles del lenguaje, esta sección se divide en dos subsecciones que son los detalles del analizador léxico y los detalles del analizador semántico, cada uno explorando y definiendo de forma mas explicita la estructura que se tomo como base para la creación del lenguaje Loop así como el lenguaje incluido en los archivos de configuración de jflex y jcup.

# Características de la aplicación

La aplicación para la compilación de los archivos ".loop" está escrita en lenguaje Java, utilizando la versión de openJDK 13.0.1 liberada el dia 15 de octubre del año 2019, utilizando como GUI Netbeans 11.2, todo esto sobre Arch Linux como sistema operativo para el desarrollo y para la prueba de la aplicación, siendo la versión de kernel Linux 5.3.8-arch1-1.

Las dependencias utilizadas como apoyo son:

- **Java-cup-11b** como versión de JCup
- **Jflex-1.7.0** como proveedor de servicios de JFlex

La aplicación se divide en tres paquetes:

- **Información:** en este encontramos las clases de error que extienden de la clase exception, además de encontrar la clase token la cual se utiliza para almacenar la información de cada token encontrado, así como contener los atributos estáticos contadorLocal y contadorGlobal los cuales tiene como función el control de los tabuladores dentro el archivo.
- **kurokitsune.main:** en esta encontramos la clase principal, las clases generadas tanto por jflex como por jcup y la clase cargar_archvio la cual extiende de JPanel y tiene la función de ser una interfaz gráfica para el proceso de selección y compilación de los archivos ".loop".
- **Tools:** aquí encontramos la clase lanzadorJFlex y lanzadorJCup las cuales tienen por función generar los archivos correspondientes al leer los archivos de configuración respectivos que se encuentran en el mismo paquete.

Los detalles relacionados al funcionamiento del código se encuentran en comentarios en dicho código, a continuación, se especifica el código que podemos encontrar agregado en el archivo jflex y jcup así como los detalles de las reglas usadas para el análisis léxico y sintáctico.

# Detalles del lenguaje

## Detalles del Analizador Léxico

Para la realización del análisis léxico se utilizó como apoyo la herramienta JFlex. Algunas cosas a tener en cuenta es que el lenguaje Loop permite la utilización de cadenas, números enteros, números reales y booleanos, además cuenta con una serie de palabras reservadas. Este lenguaje busca provocar que el usuario mantenga un control estricto en la indentación por lo que no incluye ningún indicador de fin de bloque más que el aumento de la cantidad de tabuladores para indicar que pertenece a una clase y la disminución para indicar que termina la clase. En la siguiente tabla se especifica el token y la expresión regular que se utiliza dentro del archivo flex para poder identificar cada token. Los elementos se encuentran ordenado de la misma forma tanto en la tabla como en el archivo ".flex".

| Token | Expresion regular |
|---|---|
| **FinLinea** | \r\|\n\|\r\n |
| **Tabulador_espacio** | " " |
| **Tabulador** | {Tabulador_espacio} {Tabulador_espacio} {Tabulador_espacio} {Tabulador_espacio} \| {Tabulador_espacio} {Tabulador_espacio} |
| **lineaBlanca** | [" "]+ [\r\|\n\|\r\n] |
| **String** | "\"" [0-9\|a-z\|A-Z\| "\'"\|\f\|" "\| "." \|"/"\|","]* "\"" |
| **Comentario** | "//"[0-9\|a-z\|A-Z\| "\'"\|\f\|" "\| "." \|"/"\|","]* \| "/*"[0-9\|a-z\|A-Z\| "\'"\|\f\|" "\| "." \|"/"\|","]*"*/" \| "//"[0-9\|a-z\|A-Z\| "\'"\|\f\|" "\| "." \|"/"\|","]*[\r\|\n\|\r\n] \| "/*"[0-9\|a-z\|A-Z\| "\'"\|\f\|" "\| "." \|"/"\|","]*"*/" [\r\|\n\|\r\n] |
| **r_entero** | "entero" |
| **r_real** | "real" |
| **r_booleano** | "boleano" |
| **r_cadena** | "cadena" |
| **r_nulo** | "nulo" |
| **r_escribir** | "escribir" |

| | |
|---|---|
| **r_leer** | “leer” |
| **r_devolver** | “devolver” |
| **r_cadena_entero** | “cadenaAEntero” |
| **r_cadena_real** | “cadenaAReal” |
| **r_cadena_booleano** | “cadenaABoleano” |
| **r_seno** | “seno” |
| **r_coseno** | “coseno” |
| **r_and** | “AND” |
| **r_or** | “OR” |
| **r_tangente** | “tangente” |
| **r_log** | “logaritmo” |
| **r_raiz** | “raiz” |
| **r_instanciar** | “instanciar” |
| **r_eliminar** | “eliminar” |
| **r_si** | “si” |
| **r_entonces** | “entonces” |
| **r_sino** | “sino” |
| **r_desde** | “desde” |
| **r_mientras** | “mientras” |
| **r_hacer** | “hacer” |
| **r_clase** | “clase” |
| **r_propiedades** | “propiedades” |
| **r_metodos** | “metodos” |
| **r_publicos** | “publicos” |
| **r_privados** | “privados” |
| **r_protegidos** | “protegidos” |
| **r_publicas privadas** | “publicas” |
| **r_protegidas** | “privadas” |
| **r_Principal** | “protegidas” |
| **r_constructor** | “Principal” |
| **r_extiende** | “extiende” |
| **r_incluir** | “incluir” |
| **r_comparacion** | “>” \| “<” \| “==” \| “!=” \| “<=” \| ”>=” |
| **r_puntocoma** | “;” {finlinea} \| “;” |
| **r_dospuntos** | “:” |
| **r_suma** | “+” |
| **r_resta** | “-” |
| **r_multiplicacion** | “*” |
| **r_division** | “/” |
| **r_a_parentesis** | “(” |
| **r_c_parentesis** | “)” |
| **r_igual** | “=” |
| **r_porcentaje** | “%” |

| | |
|---|---|
| **r_coma** | “,” |
| **r_abrir_c_c** | “[” |
| **r_cerrar_c_c** | “]” |
| **r_abrir_c** | “{” |
| **r_cerrar_c** | “}” |
| **r_punto** | “.” |
| **Identificador** | [a-z\|A-Z\|“_”\| “á”\| “é”\| “í”\| “ó”\| “ú”\| “ñ”\| “Ñ” \| “Á”\| “É”\| “Í”\| “Ó”\| “Ú”] [a-z \| A-Z \| 0-9 \| “_”\| “á”\| “é”\| “í”\| “ó”\| “ú”\| “ñ”\| “Ñ” \| “Á”\| “É”\| “Í”\| “Ó”\| “Ú”]* |

De esta debe ponerse principal atención en la palabra reservada “Principal” ya que esta es utilizada dentro del lenguaje para definir la clase principal y con la que debe contar todo archivo. Dentro del archivo de jflex se utilizaron dos estados para el procesamiento, el estado inicial y un “estado1”, dentro del estado inicial se omite enviar un token de fin de línea, siendo los que envían este token el punto y coma y el carácter propio de fin de línea, además de esto se inicia el “estado1” al encontrar un token distinto a fin de línea, espacio, comentario o tabulador. En el “estado1” se omite la revisión de espacios entre los comandos, además de esto el fin de línea devuelve al estado inicial y envía un token de “FINLINEA”, esto con el fin de no enviar tokens de “FINLINEA” innecesarios o que no representen nada y puedan generar un error dentro del analizador semántico, así como evitar errores surgidos a la hora de comprobar la cantidad de tabuladores dentro del léxico.

Para el control de los tabulares se utilizaron dos contadores estáticos que se encuentran dentro de la clase token, estos contadores son contadorGlobal y contadorLocal, contadorLocal es el encargado de llevar el control de la cantidad de tokens que va encontrando el analizador léxico, el cual vuelve a 0 cuando encuentra ciertos tokens, en específico al encontrar cualquier token que devuelva un “FINLINEA”, por su parte contadorGlobal mantiene el control de la cantidad de tabuladores de la última función activa, siendo un contador que sirve también para determinar errores en la indentación, los cambios en este se dan a partir del método comprobacionEspacios que se explica más adelante.

La mayoría de los tokens tienen el mismo código, cambiando únicamente en el tipo de símbolo que envía, luego varían entre estados únicamente en el hecho de que en el estado inicial llaman al estado 1 y en el “estado1” no llaman a ningún estado. La estructura general del código agregado a cada token encontrado es:

- **Estado Inicial:**

```
this.comprabacionEspacios();
tmp = new token(yycolumn, yyline, yytext(), token.contadorLocal);
yybegin(estado1);
cargar_archivo.escribirToken(tmp.toString());
```

```
return new Symbol(sym.CodigoToken, tmp);
```

- **Estado 1:**

```
this.comprabacionEspacios();
tmp = new token(yycolumn, yyline, yytext(), token.contadorLocal);
cargar_archivo.escribirToken(tmp.toString());
return new Symbol(sym.CodigoToken, tmp);
```

Como se puede apreciar en lo único que varían es en iniciar el estado 1, el método escribirToken es un método estático propio de la clase cargar_archivo que tiene por objetivo escribir el nuevo token encontrado en el archivo de tokens, este archivo lleva el mismo nombre que el archivo .loop pero con extensión .tokens y enumera todos los tokens encontrados por el analizador léxico, este archivo se guarda en el mismo sitio donde se encuentra el archivo .loop. Algunos tokens que si varían en su código son:

- **lineaBlanca:** tanto en estado 1 como en estado inicial no tiene ningún código agregado pues se ignora simplemente.
- **Finlinea y r_puntocoma:** este método varía entre los dos estados de la misma forma que los métodos anteriores, con la diferencia que en el estado1 inicia el estado inicial, pero en el estado inicial no llama a ningún otro estado. Además, otro agregado que tiene es que al encontrar cualquiera de estos dos reinicia el contadorLocal a 0, siendo parte del sistema de control de indentación del código, además ambos envían un símbolo “FINLINEA” y en vez de enviar el carácter envían un FINLINEA al token. El código agregado es:

    - **Estadoinicial:**

    ```
    token.contadorLocal = 0;
    ```

    - **Estado1:**

    ```
    tmp = new token(yycolumn, yyline, “FINLINEA”, token.contadorLocal);
    token.contadorLocal = 0;
    yybegin(YYINITIAL);
    cargar_archivo.escribirToken(tmp.toString());
    return new Symbol(sym.FINLINEA, tmp);
    ```

- **Tabulador:** Este por su parte únicamente tiene código dentro de el estado inicial, pues en este mantiene el contador de la cantidad de tabuladores que

va encontrando el analizador, en el “estado1” no tiene ningún código debido a que de aquí en adelante los espacios no son relevantes y no llevan a ninguna clase de error. Este no envía ningún tipo de token ya que el control de este se realiza a través de las variables estáticas contadorLocal y contadorGlobal. El código al encontrar un tabulador, es decir dos o cuatro espacios, es el siguiente:

- Estado inicial:

```
token.contadorLocal += 1;
```

Aparte de esto se agregan varias líneas de código, uno dentro del método comprobacionEspacios() y el segundo al encontrar el final del archivo. El código que encontramos dentro del método comprobación es el siguiente:

```
private void comprobacionEspacios(){
        if(token.contadorLocal == (token.contadorGlobal+1)){
                token.contadorGlobal += 1;
        }else if(token.contadorLocal < token.contadorGlobal){
                token.contadorGlobal = token.contadorLocal;
        }else if(token.contadorLocal == token.contadorGlobal){
        }else {
                mensaje = “Error de tabulación en:\ncol: “ + yycolumn +” || línea: “ + yyline
+ “\n”;
        Cargar_archivo.escribirNotificacion(mensaje, 1);
        }
}
```

El fin de este método es determinar la cantidad de tabuladores encontrados y llevar el control del aumento de estos, para poder determinar si lleva a un estado anormal, para la creación de este se tomaron en cuenta los siguientes puntos:

- El aumento de la cantidad de tabuladores solo puede ser de uno en uno, un aumento mayor es signo de un error en la indentación.
- La cantidad de tabuladores puede mantenerse constante.
- La cantidad de tabuladores puede disminuir desde cualquier valor hasta 0, o bajar de uno en uno sin ser en ningún caso un error.

Con esto podemos entender por que al encontrar una diferencia de uno entre el contadorLocal y el contadorGlobal lleva a aumentar en 1 el valor de contadorGlobal, al encontrar una cantidad menor dentro de contadorLocal lleva a igualar el contadorGlobal a esta cantidad y al encontrar que son iguales no se hace nada, sin embargo al encontrar una discrepancia, es decir, un aumento de más de una unidad en la cantidad de tabuladores despliega un mensaje de error dentro del text área de notificaciones a través

del método estático Cargar_archivo.escribirNotificacion, este método se explica a detalle en el código del programa.

Cuando se encuentra un fin de archivo se informa al usuario que se ha llegado al fin del mismo escribiendo una notificación en el text área respectivo, esto a través del método estatico cargar_archivo.escribirNotificaion(), luego devolvemos un sym.EOF a; analizador sintáctico.

```
mensaje = “Finalizacion de análisis lexico\n”;
cargar_archivo.escribirNotificacion(mensaje, 0);
return new Symbol(sym.EOF);
```

## Detalles del Analizador Semántico

Para el analizador semántico se tomó en cuenta una serie de restricciones del lenguaje, así como ciertas características que marcan su estructura. Algunas cosas importantes a tener en cuenta es que una instrucción puede acabar o bien en un fin de línea o bien con un punto y coma, pero siempre debe encontrarse toda la instrucción en la misma línea, de lo contrario se tomara como instrucciones distintas, esto le da una gran importancia a la identación, esta debe llevarse a cabo usando o bien dos espacios en blanco o bien un tabulador, con cada aumento de la cantidad de tabuladores se indica que el código pertenece a la sección que tiene el tabulador_actual – 1. El lenguaje permite la utilización de números enteros, reales, cadenas, booleanos y nulos. Siendo los datos nulos únicamente utilizados para indicar que un método no devuelve ningún dato.

Los identificadores por otra parte no pueden iniciar con números, sin embargo, dentro del identificador si pueden incluir números y algunos caracteres especiales. Por su parte el trabajo con enteros permite su asignación a variables reales, cadenas y booleanos, de la misma forma un real se puede asignar a un entero, una cadena o un booleano. Un booleano también puede guardarse dentro de un tipo cadena, un tipo real o entero. Los comentarios se encontraran delimitados por /* comentario */ y por //. En lo que respecta a operaciones se pueden realizar asignaciones de variables a través del carácter “=” como en cualquier lenguaje de alto nivel, las operaciones aritméticas que se permiten son suma (+), resta (-), multiplicación (*), división (/), modulo (%), exponenciación (^). De estos las cadenas únicamente podrá utilizar el operador “+” para concatenar texto. Adams de permitir el pre y post incremento. En lo que respecta a operaciones lógicos se pueden utilizar las operaciones de AND y OR sobre tipos booleanos. Para agrupación se tendrá los paréntesis normales.

Para la entrada y salida se tendrá las palabras reservadas “escribir” y “leer”, escribir podrá recibir parámetros a través de cadenas, valores o cadenas de texto, en el caso de leer deberá recibir directamente los valores dentro de una variable. Para las estructuras selectivas se tendrá varias formas un “si” que funcionara como un if y

comparaciones directas a través de los signos de comparación, pudiendo usarse solamente mayor que (>), menor que (<), igual que (==) y distinto que ( !=). En el caso de las comparaciones directas deberán recibirse dentro de una variable tipo booleano. En el caso de las funciones iterativas se tendrán varias opciones, un desde que funciona como un for, recibiendo una condición a través de la palabra reservada “mientras” y finalizando con hacer recibiendo luego el código respectivo. Luego tenemos un do while que empieza con la palabra reservada hacer, luego recibe las instrucciones y finalmente pone la condición a través de la palabra reservada “mientras”. Además de esto se podrán tener funciones sueltas, estas no pertenecen a ninguna clase y siempre serán funciones públicas, su estructura básica debe cumplir con las mismas reglas de las funciones de las clases regulares. Las funciones deben tener una declaración donde se aclare el tipo de dato devuelto, el identificador de la función y los argumentos. Además, debe poder aplicarse polimorfismo sobre las funciones. El lenguaje lleva varias funciones especiales ya definidas como pueden ser la conversión de una cadena a entero, real y booleano a través de la función cadenaAEntero, cadenaAReal y cadenaABoleano, estas reciben de parámetro una variable tipo cadena. También existen funciones matemáticas como seno, coseno, tangente, logaritmo y raíz, las cuales reciben como parámetro una variable tipo real.

En el caso de las clases esta debe indicar que son una clase a través de la palabra reservada “clase” seguidas del identificador de la clase, además pueden heredar de clases padre, lo que se indicara con la palabra reservada “extiende” seguida de el nombre de la clase padre. Además de esto una clase cuenta con una serie de secciones como es una sección para las propiedades y una sección para los métodos, siendo cada uno definido como propiedades públicas, propiedades privadas y propiedades protegidas. Con los métodos se define de la misma manera con la excepción que en vez de estar en femenino están en masculino, es decir públicos, privados y protegidos. Dentro de las distintas secciones de propiedades se pueden definir las propiedades de clase, mientras tanto dentro de los métodos se pueden definir variables propias de la clase, pudiéndose inicializar en esa misma instrucción. Por su parte los métodos aparte de su declaración contiene el código propio luego de la declaración.

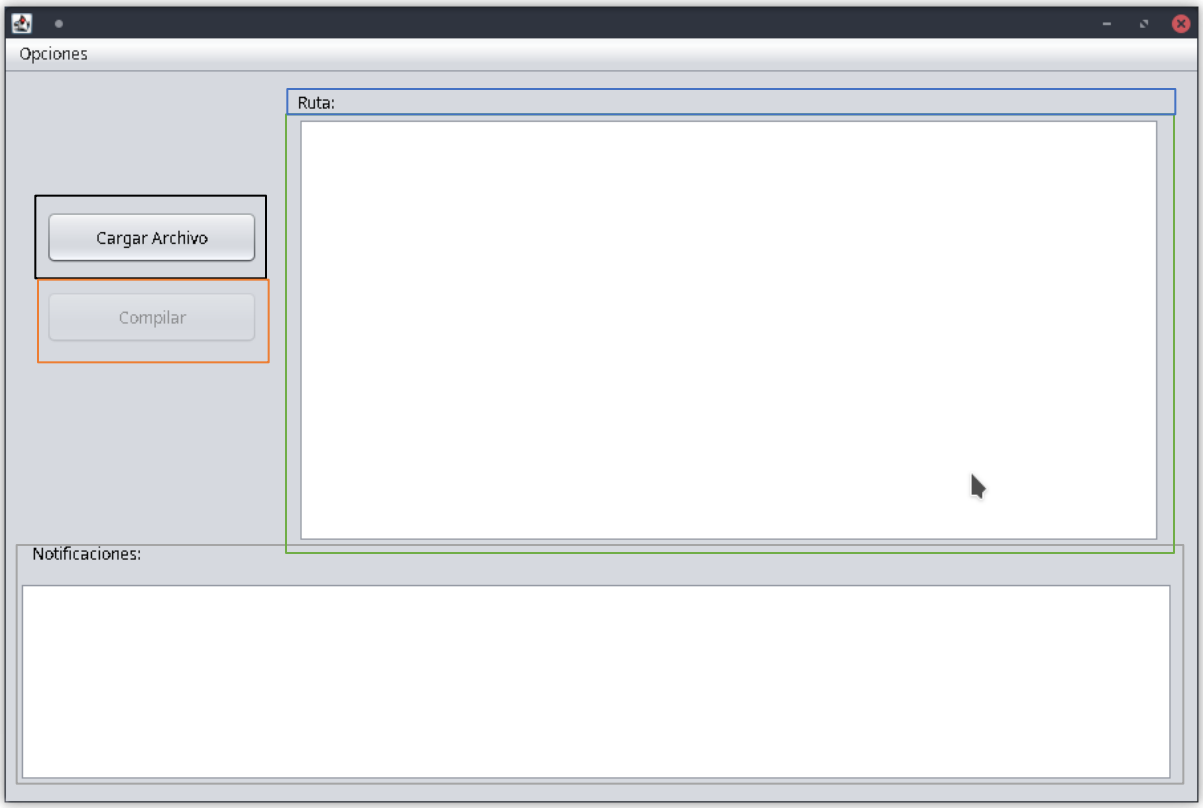
La clase principal siempre será “entero Principal()”, dentro de esta se podrá instanciar otras clases a través de la palabra reservada “instanciar” seguida del nombre de la clase, esto funcionando igual que en cualquier otro lenguaje como por ejemplo Java, pero omitiendo en este caso la palabra new. Además, dentro de las clases pueden existir tanto constructores como destructores que ayuden a eliminar variables que ya no se usen. Al inicio de todo archivo loop está la opción de incluir librerías o archivos .loop externas a través de la palabra “incluir” acompañada de una cadena que indique la localización del archivo en cuestión. Para la inclusión y uso de arreglos se indica con dos corchetes cuadrados junto al identificador de la variable la cantidad de espacios que puede tener (variable [0] o bien variable[]), además de poder inicializar el arreglo a través del carácter “=” y {valor1, valor2, …, valori}.

# Manual de Usuario

# Uso de la aplicación

En esta sección se detalla como interactuar con la aplicación, como se ha mencionado en la sección técnica la aplicación está escrita en Java y esta ideada para funcionar sobre sistemas operativos Linux. Actualmente se tiene una sola funcion habilitada que es la de cargar un archivo en extensión de Loop para compilarlo y generar un archivo de salida en .asm. La pantalla principal es la siguiente, como se puede observar se cuenta con una pestaña con un menú que al desplegarse despliega las opciones disponibles.

Como puede observarse en la imagen anterior la única opción disponible es la de compilar un archivo, al hacer clic muestra el formulario con el que debemos interactuar, este cuenta con una serie de elementos que se describen a continuación.

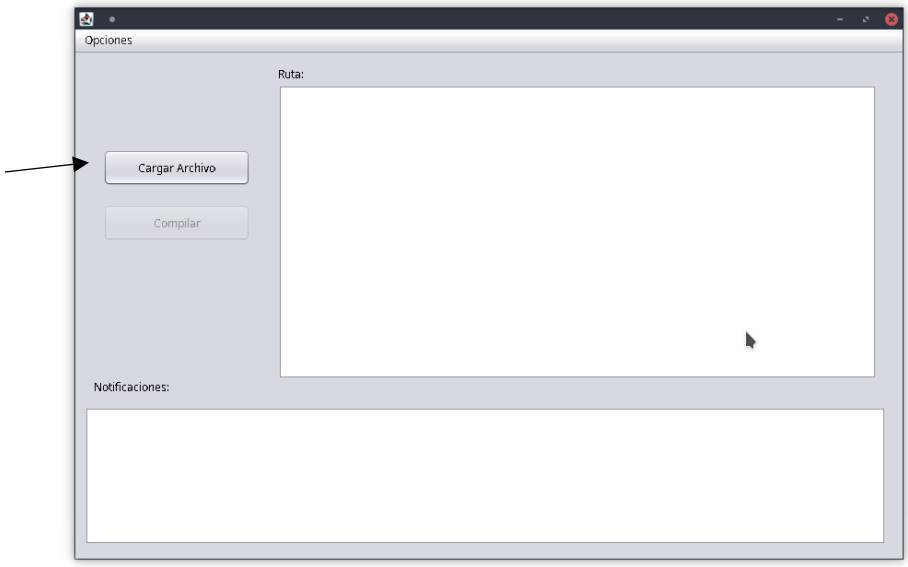



- Botón para cargar un archivo, este lleva por titulo “cargar archivo” al hacer click encima se despliega un selector de archivos que muestra únicamente directorios y los archivos “.loop”.
- Botón para compilar el archivo, este en un principio se encuentra desactivado, pero al seleccionar un archivo por primera vez se activa, permitiendo compilar dicho archivo.

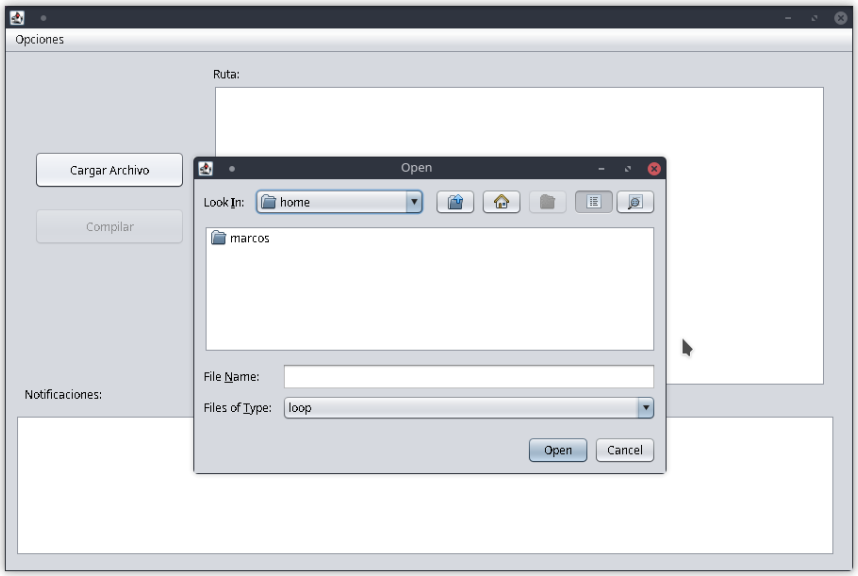
- Etiqueta de ruta, la cual despliega la ruta del archivo seleccionado para poder identificar el archivo sobre el que trabajara el compilador.
- Área de texto que muestra el contenido del archivo al momento de seleccionar un archivo, cada vez que se seleccione un archivo nuevo esta mostrara el contenido de dicho archivo.
- Área de texto que muestra las notificaciones que genera el compilador durante su proceso, en caso de no encontrar ningún error notificara esto en esta área, pero si encuentra algún problema también notificara de estos problemas en esta sección.

El proceso para utilizar esta función será el siguiente:

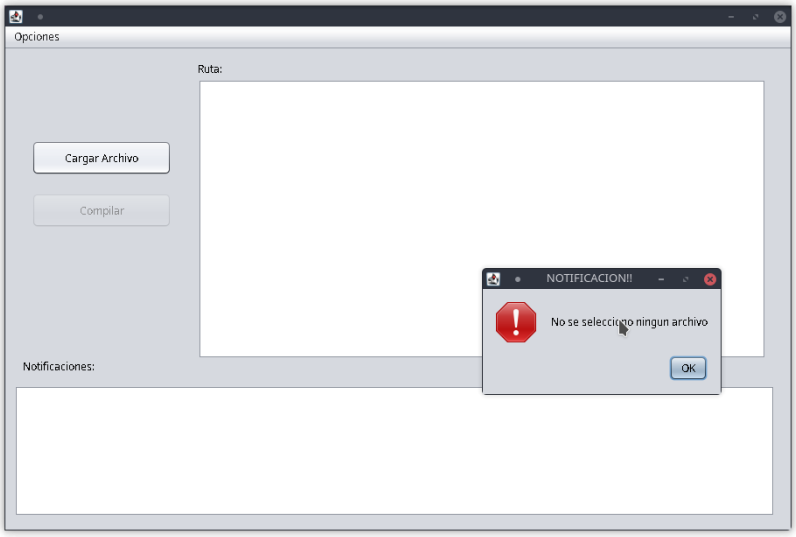
1. Lo primero que debemos hacer luego de seleccionar la opción de “compilar archivo” en la pantalla principal debe ser seleccionar el archivo que deseamos compilar, para esto utilizaremos el botón que tiene por identificador el texto “Seleccionar Archivo”



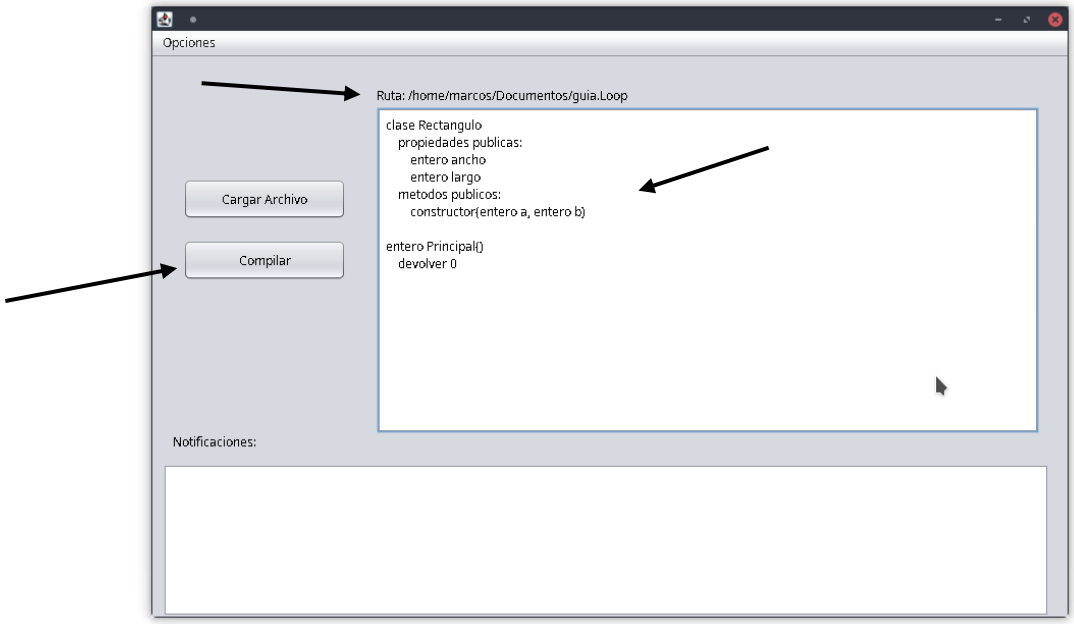

lo cual desplegará un selector de archivos que nos permitirá seleccionar el archivo correspondiente. Debemos tener en cuenta que este solo muestra las carpetas y los archivos con extensión “.loop”, “.Loop” y “.LOOP”. Por defecto este abre la ruta “/home/” automáticamente.




En caso cancelemos la operación se mostrará una advertencia que nos indicará que no hemos seleccionado ningún archivo.

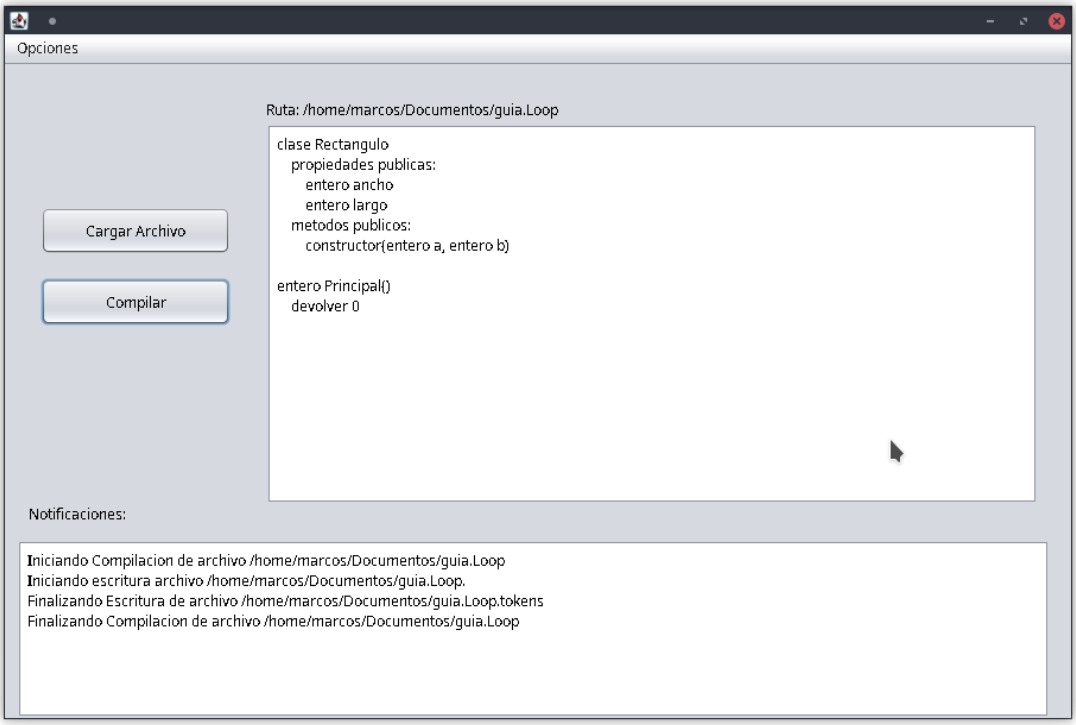



2. Finalmente, si seleccionamos un archivo y hacemos clic sobre el botón de “open” entonces veremos que en el cuadro de texto que tenemos del lado derecho de la interfaz grafica se despliega el contenido del archivo, además arriba de este se muestra la ruta en la que se encuentra el archivo.




Debemos observar también que el botón de “compilar” se activara y que el área de notificaciones se limpiara automáticamente, ya que solo muestra las notificaciones del último proceso. Una vez verificado que es el archivo que deseamos compilar pasamos al siguiente paso.

3. Ahora procedemos a hacer clic sobre el botón de compilar, esto iniciara el proceso.



Como podremos ver se notifica al usuario del inicio del proceso de compilación del archivo, así como también del inicio de la escritura de un archivo de extensión .tokens que se ubica en el mismo directorio donde esta el archivo .loop. El archivo de extensión tokens muestra todos los tokens encontrados dentro del archivo, así como otra información útil. En caso de existir problemas o errores en el proceso de compilación estos se desplegarán en el área de notificaciones indicando en que línea se dio el problema y que tipo de problema fue, si no encuentra ningún error se notificara del fin de la compilación en esta misma área.

Algunas cosas para tener en cuenta es que en caso de querer seleccionar otro archivo y cancelar en último momento sin haber seleccionado el archivo, se compilara el ultimo archivo seleccionado al compilar el archivo.

# Detalles del Lenguaje

A continuación, se describe información concerniente al archivo de Loop y del lenguaje a utilizar, dando información útil al usuario de las herramientas disponibles a la hora de crear un programa en este lenguaje, así como la estructura correcta que este debe llevar

## Información del archivo

El archivo debe escribirse con extensión “.loop”, “.Loop”, “.LOOP” ya que estas son las extensiones que reconoce el compilador al momento de abrir un archivo nuevo. El archivo puede ser editado en cualquier tipo de editor de texto.

## Palabras y caracteres reservados

Como en todo lenguaje se tiene una serie de palabras reservadas y caracteres que tienen utilidades específicas para el lenguaje, estas se describen en la siguiente tabla, describiendo en la primera columna el carácter o palabra reservada y en la segunda su funcionalidad.

| Palabra/carácter | Función |
|---|---|
| **Fin de Línea** | Al momento de dar un enter, se genera un fin de línea, para el lenguaje loop este carácter marca el final de una instrucción. |
| **Tabulador** | El lenguaje loop tiene un control estricto sobre la indentación del código, por lo que |

| | |
|---|---|
| | el tabulador juega un papel fundamental, la forma de escribir el lenguaje es la siguiente:<br>• Si la cantidad de tabuladores es + 1 que el de la última función, método, clase o inicio de sección significa que pertenece a esta sección.<br>• Si la cantidad es igual al de la línea anterior significa que todas pertenecen a la misma clase, método o sección padre.<br>• Si es menor que el anterior significa que se cierra los métodos, clases o funciones anteriores. |
| **Espacio en Blanco** | El espacio en blanco por si solo es ignorado, pero si se colocan dos seguidos se toma en cuenta como un tabulador. |
| **Línea en Blanco** | Las líneas en blanco, es decir toda línea que no tenga ningún carácter mas que un fin de línea o bien que solo tenga espacios en blanco, son ignoradas y no se toman en cuenta. |
| **“cadena”** | La estructura para definir una cadena es encerrar entre dos comillas dobles el texto que desea utilizarse, por ejemplo:<br>“esto es una cadena de ejemplo” |
| **Comentario** | Los comentarios pueden definirse de dos maneras, utilizando doble barra (//) o utilizando (/* */) por ejemplo:<br>// Esto es un comentario<br>/*<br>Esto también es un comentario<br>*/ |
| **Tipos de datos** | |
| **entero** | |

| | |
|---|---|
| | Esta palabra al utilizarla antes de un identificador indica que la variable es de tipo entero, en caso de usarlo en la declaración de un método indica que el valor a ser devuelto es de tipo entero. |
| **real** | Esta palabra al utilizarla antes de un identificador indica que la variable es de tipo real, en caso de usarlo en la declaración de un método indica que el valor a ser devuelto es de tipo real. |
| **boleano** | Esta palabra al utilizarla antes de un identificador indica que la variable es de tipo booleano, en caso de usarlo en la declaración de un método indica que el valor a ser devuelto es de tipo booleano. |
| **cadena** | Esta palabra al utilizarla antes de un identificador indica que la variable es de tipo cadena. |
| **nulo** | Este tipo solo puede utilizarse al declarar un método ya que indica que dicho método no devuelve ningún tipo de dato. |
| **Instrucciones** | |
| **escribir** | Esta instrucción se utiliza para escribir en pantalla una cadena indicada, a esta instrucción se le pueden pasar objetos tipo cadena, enteros, reales o booleanos, la estructura debe ser como la siguiente:<br>escribir “cadena”<br>escribir nombreVariable |
| **leer** | Esta instrucción permite leer datos de un dispositivo de entrada, el parámetro que recibe es la variable que contendrá el dato:<br>leer nombreVariable |

| | |
|---|---|
| **devolver** | Esta instrucción se utiliza para indicar el valor que será devuelto por una funcion o método, va acompañada del valor o de la variable que contiene el dato a ser regresado, este debe coincidir con el tipo declarado en la definición del método.<br>devolver nombreVariable |
| **instanciar** | Esta palabra se utiliza a la hora de crear una instanciación de una clase, la estructura con la que debe cumplir al momento de utilizarse es:<br>NombreClase nombreVariable = instanciar nombreClase() |
| **eliminar** | Este se utiliza para eliminar una variable. La estructura con la que debe cumplir es:<br>eliminar nombrevariable |
| **si** | Es el equivalente de los if en otros lenguajes, luego de este va la condicionante, la estructura que lleva es<br>si condición entonces<br>---código-- |
| **entonces** | Forma parte de la sentencia condicionante si, este se coloca después de la condición<br>si condición entonces<br>---código-- |
| **sino** | Es el equivalente de else en otros lenguajes, se coloca luego de el primer conjunto de instrucciones de un si, es opcional su colocación<br>--codigo—<br>sino<br>--codigo— |

| | |
|---|---|
| **desde** | Este inicia un ciclo iterativo, es el equivalente de for de otros lenguajes, la estructura que debe cumplir es:<br>desde <variable\|inicialización> mientras <condición> [incrementar\|decrementar <numero>] hacer |
| **mientras** | Como se ve en la estructura de desde, forma parte de la estructura iterativa. |
| **hacer** | Es la parte final de la declaración de una estructura iterativa desde, la estructura se especifica en la descripción de la palabra reservada “desde”. |
| **clase** | Esta palabra reservada se utiliza a la hora de declarar una clase, la estructura que debe cumplir es:<br>clase identificador [extiende identificadorPadre] |
| **propiedades** | Indica que todas las líneas siguientes de código son declaración de propiedades de la clase, este va acompañado del tipo de acceso permitido<br>Propiedades <acceso>: |
| **metodos** | Esta palabra reservada indica que de aquí en adelante se crearan métodos, la estructura que debe cumplir es:<br>metodos <acceso> |
| **publicos** | Este indicador de acceso, indica que todos los métodos que siguen a la sección correspondiente son de acceso publico |
| **privados** | |

| | |
|---|---|
| | Este indicador de acceso, indica que todos los métodos que siguen a la sección correspondiente son de acceso privado |
| **protegidos** | Este indicador de acceso, indica que todos los métodos que siguen a la sección correspondiente son de acceso protegido. |
| **publicas** | Este indicador de acceso, indica que todas las propiedades que siguen a la sección correspondiente son de acceso publico |
| **privadas** | Este indicador de acceso, indica que todas las propiedades que siguen a la sección correspondiente son de acceso privado |
| **protegidas** | Este indicador de acceso, indica que todas las propiedades que siguen a la sección correspondiente son de acceso protegido |
| **Principal** | Esta palabra está reservada para la declaración del método principal, este método puede o no recibir parámetros, este debe cumplir con la siguiente estructura: entero Principal() |
| **constructor** | Esta palabra reservada se utiliza para declarar los constructores de las distintas clases, estos métodos siempre deben ser públicos y siguiendo la siguiente estructura: constructor(argumentos) |
| **extiende** | Como se vio en la estructura de declaración de una clase, esta palabra se utiliza para indicar que una clase hereda de otra, siendo acompañada por el identificador de dicha clase. |

| | |
|---|---|
| **incluir** | Esta palabra se utiliza para indicar que se desea importar otros archivos .loop para utilizarlos dentro del codigo la estructura que sigue es:<br>Incluir “/ruta/a/el/archivo” |
| **Funciones Incluidas** | |
| **cadenaAEntero** | Esta función recibe una cadena y devuelve su valor en tipo entero, la estructura que debe mantener es:<br>entero cadenaAEntero(cadena variableC) |
| **cadenaAReal** | Esta función recibe una cadena y devuelve su valor en tipo entero, la estructura que debe mantener es:<br>entero cadenaAReal(cadena variableC) |
| **cadenaABoleano** | Esta función recibe una cadena y devuelve su valor en tipo boleano, la estructura que debe mantener es:<br>boleano cadenaABoleano(cadena variableC) |
| **seno** | Esta función tiene por objetivo calcular el seno de un numero tipo real que recibe como parámetro, devolviendo un valor real.<br>real seno(real n) |
| **coseno** | Esta función tiene por objetivo calcular el coseno de un numero tipo real que recibe como parámetro, devolviendo un valor real.<br>real coseno(real n) |
| **tangente** | Esta función tiene por objetivo calcular el tangente de un numero tipo real que recibe como parámetro, devolviendo un valor real. |

| | |
|---|---|
| | real tangente(real n) |
| **logaritmo** | Esta función tiene por objetivo calcular el logaritmo base 10 de un numero tipo real que recibe como parámetro, devolviendo un valor real.<br>real logaritmo(real n) |
| **raiz** | Esta función tiene por objetivo calcular la raíz cuadrada de un numero tipo real que recibe como parámetro, devolviendo un valor real.<br>real raiz(real n) |
| **>** | Este operador es un operador de comparación lógico, se utiliza para determinar si un valor1 es mayor que un valor 2, devuelve un boleano<br>Valor1>Valor2<br>Variable1>variable2<br>Varaible1>valor1<br>Valor1 > variable1 |
| **<** | Este operador es un operador de comparación lógico, se utiliza para determinar si un valor1 es menor que un valor 2, devuelve un boleano<br>Valor1<Valor2<br>Variable1<variable2<br>Varaible1<valor1<br>Valor1 < variable1 |
| **==** | Este operador es un operador de comparación lógico, se utiliza para determinar si un valor1 es mayor que un valor 2, devuelve un boleano<br>Valor1 == Valor2<br>Variable1 == variable2<br>Varaible1 == valor1 |

| | |
|---|---|
| | Valor1 == variable1 |
| **!=** | Este operador es un operador de comparación lógico, se utiliza para determinar si un valor1 es distinto a un valor 2, devuelve un boleano<br>Valor1 != Valor2<br>Variable1 != variable2<br>Varaible1 != valor1<br>Valor1 != variable1 |
| **;** | Este carácter se utiliza para señalar un fin de línea al colocarlo al final de una instrucción, no es obligatorio para este fin, pero siempre es una opción. |
| **:** | Carácter utilizado para señalar el inicio de la sección de declaración de propiedades o de métodos. |
| **AND** | Este operador lógico es utilizado para realizar una operación a nivel de bits en una variable dada, siendo la sintaxis<br>Variable1 AND variable2 |
| **OR** | Este operador lógico es utilizado para realizar una operación a nivel de bits en una variable dada, siendo la sintaxis<br>Variable1 OR variable2 |
| **+** | Este operador se utiliza para realizar la suma de dos valores, ya sea enteros o reales. En el caso de strings se utiliza para la concatenación de valores.<br>Variable1 + variable2<br>Dato1 + variable2<br>Vairable1 + dato2<br>Dato1 + dato2<br>Cadena1 + dato\|cadena\|variable |

| | |
|---|---|
| - | Este operador se utiliza para realizar la resta de dos valores, ya sea enteros o reales.<br>Variable1 - variable2<br>Dato1 - variable2<br>Vairable1 - dato2<br>Dato1 - dato2 |
| * | Este operador se utiliza para realizar la multiplicación de dos valores, ya sea enteros o reales.<br>Variable1 * variable2<br>Dato1 * variable2<br>Vairable1 * dato2<br>Dato1 * dato2 |
| / | Este operador se utiliza para realizar la división de dos valores, ya sea enteros o reales.<br>Variable1 / variable2<br>Dato1 / variable2<br>Vairable1 / dato2<br>Dato1 / dato2 |
| ( | Se utiliza para la agrupación de datos, en el caso de métodos se utiliza para indicar que lo que sigue son los parámetros del método. |
| ) | Se utiliza para cerrar las agrupaciones del ultimo paréntesis abierto. |
| = | Se utiliza para la asignación de datos<br>Variable = dato |
| % | Indica una operación de modulo, entre dos operadores, siguiendo la estructura de los operadores normales. |

| | |
|---|---|
| , | Se utiliza para la separación de los valores en los arreglos. |
| [ | Se utiliza para indicar que una variable es un arreglo y la cantidad de espacios con los que cuenta el arreglo. |
| ] | Cierra el ultimo corchete cuadrado abierto. |
| { | Se utiliza para la inicialización de datos de un array, dentro se coloca la lista de datos a ingresar en el array. |
| } | Cierra el ultimo corchete abierto |
| . | Se utiliza durante con objetos para indicar la propiedad o método del objeto instanciado al que se desea acceder. |
| **Identificador** | Los identificadores tienen la característica de no poder empezar con valores numéricos, además no permite la utilización de caracteres especiales como guion bajo y ñ tanto en mayúscula como en minúscula. |

## Estructura de un programa en Loop

Una vez comprendido todas las herramientas y palabras reservadas que ofrece el lenguaje, podemos comprender como estructurar correctamente un programa en dicho lenguaje, recordemos que este es un lenguaje orientado a objetos, por lo tanto, se permite la creación de clases que pueden tener métodos y atributos. La estructura básica que tenemos es la siguiente:

```
incluir " ../../ impresion . loop "
incluir "fubnlectura. loop "
clase Impar
        propiedades publicas :
                cadena hola
        metodos publicos :
                entero esImpar ( entero numero )
                        si numero % 2 entonces
                                devolver 1
                        sino
                                devolver 0

clase HolaMundo
        propiedades privadas:
                entero a
                cadena nombre
                Impar i
        metodos publicos :
                constructor ()
                        i = nuevo Impar ()
                        a = 0
                        nombre = ""

                entero inicio ()
                        escribir " escriba su nombre "
                        leer nombre ;
                        escribir " escriba un valor entre 1 y 30"
                        // se guarda un numero en la variable a
                        leer a
                        /* escribe en pantalla la constante junto
                        * al valor de la variable nombre
                        */
                        escribir " Hola ", nombre
                        si i. esImpar (a) entonces
                                escribir "El numero que eligio es impar "
                        sino
                                escribir "El numero que eligio es par "
                        devolver 0
entero sumar ( entero a, entero b)
`       devolver a + b

entero Principal ()
        HolaMundo hola = instanciar HolaMundo ()
        hola . inicio ()
        devolver 0
```

Las tres secciones que podemos identificar fácilmente son:

- **Sección de “incluir”:** La sección para importar librarías de otros archivos “.loop”, esta se encuentra al inicio del archivo donde cada instrucción puede se encuentra en una sola línea, en el ejemplo la encontramos resaltada en color verde.
- **Sección de Funciones Sueltas y clases:** aunque mas adelante se describe mejor la forma en que está estructurada cada subsección, a grandes rasgos en esta sección se pueden declarar clases con sus respectivos métodos y propiedades, así como funciones sueltas, las cuales siempre son publicas y pueden ser accedidas por cualquier clase. Esta sección esta resaltada en naranja
- **Sección del código principal:** esta sección tiene el código base de la aplicación, funciona como cualquier función o método con la diferencia de que esta siempre será entera y llevará por identificador “Principal”, esta resaltada en azul.

## *Estructura de instrucción “incluir”*

Un mismo programa escrito en “.loop” puede incluir varias librarías, esto a través de colocar varias instrucciones seguidas en la correspondiente sección. Para indicar que queremos incluir una librería utilizaremos la palabra reservada “*incluir”* luego de esto se indicara la dirección del archivo que queremos incluir, debe tenerse en cuenta que solo se permite dos carpetas por encima de la carpeta actual.

```
Incluir “../../nombreArchivo.loop”
```

## *Estructura de una clase*

Como todo lenguaje orientado a objetos, las clases que incluyamos pueden o no heredar de otras clases, así como sus métodos pueden tener polimorfismo, para indicar que queremos crear una clase nueva utilizamos primero la palabra reservada “*class”* seguido indicamos el nombre que queremos darle a la clase, esto siguiendo las mismas indicaciones que para cualquier identificador, no puede iniciar con números, no puede usar caracteres especiales, etc.

```
clase identificadorClase
```

Para indicar que la clase hereda de otra clase se utiliza la palabra reservada “*extiende”* luego de indicar el nombre de la clase, seguido de esta palabra se indica el nombre de la clase padre.

```
clase identificadorClase extiende identificadorClasePadre
```

## *Estructura de una función suelta*

Una función suelta lleva la misma estructura que cualquier método, primero debemos indicar el tipo de dato que este devuelve, siendo nulo en caso de no devolver ningún tipo de dato. Luego se indica el identificador de la función seguido por dos paréntesis.

tipoDato identificador()

En caso de recibir parámetros estos se incluirán entre los paréntesis separados por comas.

tipoDato identificador(tipo param1, tipo param2)

Una vez contemplado esto, se agregan las líneas de código pertenecientes a la función con al menos + 1 tabuladores de la cantidad de tabulaciones de la función.

## *Estructura Propiedades*

La sección de propiedades es la primera en ser incluida en una clase, esta tiene tres variantes, cada una depende del nivel de acceso que se tenga a las propiedades pudiendo ser esta privadas, protegidas o bien públicas. Para inicializar esta subsección empezamos por poner la palabra reservada "*propiedades"* seguida del identificador de acceso y finalizando con dos puntos:

propiedades identificadorAcceso:

Luego se incluirán las propiedades indicando el tipo de dato y luego el identificador de la variable.

tipoDato Identificador

entero Largo

## *Estructura de un método*

Debe tenerse en cuenta que estas estructuras o subsecciones deben ubicarse luego Los métodos deben incluirse en la sección respectiva, a diferencia de lenguajes como Java, donde se indica el nivel de acceso del método durante la declaración de este, en Loop se tiene una sección para métodos públicos, privados y protegidos, ninguna de esta es obligatoria y puede incluirse cualquiera de las tres, dos de ellas o todas. Para indicar esto se lleva la siguiente estructura:

metodos identificadorAcceso:

Con esto indicaremos que iniciamos la sección de declaración de métodos, métodos que tienen la restricción de acceso indicada, de la misma forma que sucede con las funciones sueltas para crear un método nuevo primero indicamos el tipo de dato que

devuelve, luego procedemos a asignarle un identificador y finalmente los parámetros o argumentos que recibe.

tipoDato identificador(tipo param1, tipo param2)

Dentro de la sección de métodos debe incluirse los constructores y destructores de las clases. El constructor y el destructor funcionan de la misma forma que cualquier otro método, con la diferencia que estos no devuelven ningún dato y siempre llevan por identificador la palabra reservada "*constructor"* o "*destructor"* siendo su estructura como sigue:

constructor(tipo param1)

…código…

destructor(tipo param1)

…código…

## *Estructura del programa principal*

El programa principal como toda función inicia por indicar el tipo de dato que devuelve, que en este caso siempre será entero, luego se indica el identificador que siempre debe ser la palabra reservada "Principal" y finalmente los argumentos que recibe.

Entero Principal()

Una vez hecho esto se debe incluir el código de la función principal.