Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de lenguajes y compiladores 2
Vacaciones de diciembre 2018
Catedrático: Ing. Kevin Lajpop
Tutor académico: Javier Navarro

Coline Teacher

Segundo proyecto de laboratorio

Contenido

Índice de tablas

Índice de imágenes

1 Descripción

Para el segundo proyecto del laboratorio se deberá desarrollar una aplicación de escritorio que simule una aplicación web, la cual permitirá a las personas que no han tenido ninguna experiencia con lenguajes de programación, que puedan aprender a programar de una forma fácil e intuitiva. Esta aplicación tendrá el nombre de Coline Teacher. Esta aplicación deberá ser desarrollarla e implementarla con el lenguaje C++ utilizando el framework Qt.

La aplicación de Coline Teacher deberá implementar el siguiente lenguaje:

- Coline: está basado en el lenguaje C++. Este será un tipo de lenguaje orientado a objetos.

En esta aplicación será posible publicar lecciones que ayuden al usuario en el aprendizaje del lenguaje. Existirán dos tipos de lecciones, el primero será Dummy-coach que servirá para aprender la sintaxis de Coline y su respectivo funcionamiento, el segundo será el tipo de lección Pro-coach que servirá para poder desarrollar algoritmos y solucionar problemas a través del lenguaje Coline.

Los usuarios del sistema podrán crear nuevas lecciones y participar en las que se encuentren creadas. Se podrá subir tareas como parte de las lecciones para evaluar el aprendizaje de los usuarios y deberán de cargar el resultado esperado para que la calificación pueda ser automática.

La aplicación implementará un debugger que ayude a los usuarios a comprender los ejemplos dejados en las lecciones.

2 Objetivos

2.1 Objetivo General

- Aplicar los conocimientos de Organización de Lenguajes y Compiladores 2 en la creación de una solución de software.

2.2 Objetivos Específicos

- Utilizar herramientas web específicas para la creación de un compilador que cumpla efectivamente con los requerimientos de un lenguaje predeterminado.
- Aplicar los conocimientos de generación de código intermedio en formato de cuádruplos en la implementación de una herramienta que permita generar dicho código a partir de un lenguaje de alto nivel.

3 Coline Teacher

Coline Teacher, será una aplicación que tendrá un conjunto de herramientas que ayudará a los usuarios a poder programar de manera intuitiva y fácil. Esta aplicación implementará un lenguaje de fácil aprendizaje. Además, estará enfocado en personas cuyo idioma sea el español.

En la aplicación Coline Teacher será posible implementar la traducción de código de alto nivel a código intermedio y ejecución de código intermedio. Este proceso se llevará a cabo por medio de dos formas, la compilación que pasará por todo el proceso descrito y mostrará el resultado y el debugger que mostrará paso a paso el proceso de compilación con el fin de facilitar el aprendizaje.

La aplicación aceptará la publicación de lecciones a través de ella, y les dará acceso a los usuarios a visualizar el contenido de las lecciones y a participar en las tareas de las lecciones.

3.1 Lecciones

Las lecciones serán pequeñas explicaciones que buscan enseñar a los usuarios a programar. Existirán dos tipos de lecciones que se describen a continuación.

Dummy-coach: este tipo de lección se enfocará en enseñar al usuario a usar el lenguaje Coline. Esto quiere decir que, las lecciones que pertenezcan a este tipo se enfocarán a explicar el funcionamiento de las sentencias del lenguaje Coline y la sintaxis correcta de cada sentencia.

Pro-coach: será un tipo de lección para aprender y mejorar las habilidades de resolución de problemas por medio del lenguaje Coline. En este tipo de lección se enseñará al usuario a implementar algoritmos que den solución a determinados problemas.

Una lección, independientemente el tipo que sea, estará compuesto por:

- **Título:** nombre que identificará a la lección.
- **Explicación:** parte textual de la lección. Contiene una explicación del componente o un enunciado del problema al que se dará solución.
- **Código de ejemplo:** ejemplo de código de alto nivel que mostrará el funcionamiento que se estará enseñando en la lección.
- **Enunciado de tarea:** se podrá subir, de manera opcional, un enunciado de una tarea para que los usuarios puedan evaluar el aprendizaje.
- **Pruebas:** si se publica una tarea, deberá subir de forma obligatoria una o varias pruebas que califiquen automáticamente la tarea. Cada prueba deberá tener los datos de entrada y la salida esperada.

3.2 Funcionalidades de la aplicación

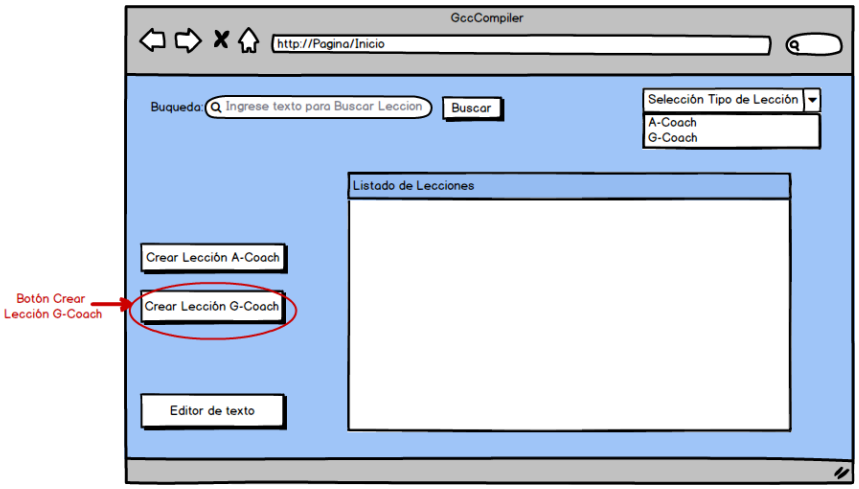
Dentro de la aplicación se podrán realizar las funcionalidades que se describen a continuación.

3.2.1 Crear una nueva lección de tipo Dummy-coach

En este caso se desea crear una nueva lección en la aplicación. La lección se enseñará el funcionamiento de la sentencia “Si” y la forma correcta de usarlo en el lenguaje Coline. Esta lección pertenecerá al tipo Dummy-coach. Para poder crear esta nueva lección se deberán seguir los siguientes pasos.

1. Ingresar a la aplicación. Se mostrará la pantalla de inicio de la aplicación y se deberá hacer clic en el botón “Crear lección Dummy-coach” como se muestra en la imagen 1.

Imagen 1: pantalla de inicio para crear una nueva lección Dummy-coach.



2. Se mostrará la pantalla para el ingreso de una nueva lección de tipo Dummy-coach. En esta pantalla se ingresará una lección sobre la sentencia “Si”. La información que se ingresará en la lección será:
   - **Título:** Si.
   - **Explicación:** esta sentencia evaluará una condición que determinará si se ejecuta el grupo de instrucciones que corresponden a de verdad de la condición o al grupo de instrucciones que corresponden a la verdaderodad de la condición.
   - **Código de ejemplo:**

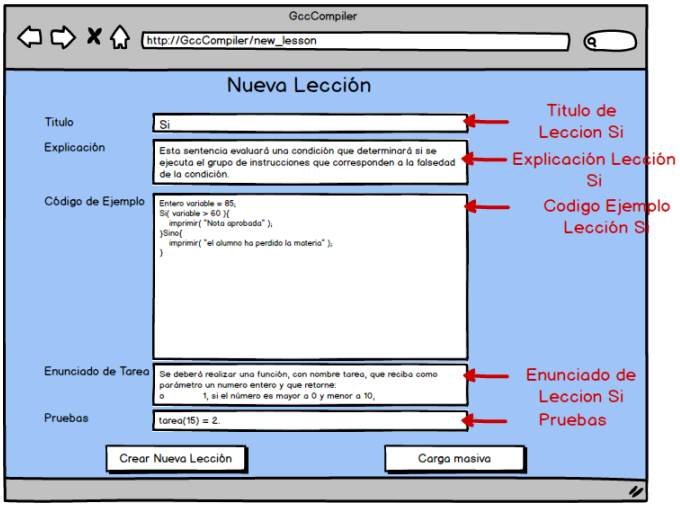
```
Entero variable = 85;
Si( variable > 60 ){
    imprimir( “Nota aprobada” );
}Sino{
    imprimir( “el alumno ha perdido la materia” );
}
```

- **Enunciado de tarea:** se deberá realizar una función, con nombre tarea, que reciba como parámetro un número entero y que retorne:
    - 1, si el número es mayor a 0 y menor a 10,
    - 2, si el número es mayor o igual a 10 y menor a 20.
    - 3, si el número es mayor o igual a 20.
- **Pruebas:** tarea(15) = 2.

Nota: tomar en cuenta que la prueba es una llamada a la función, que realizará el estudiante y se iguala al valor esperado.

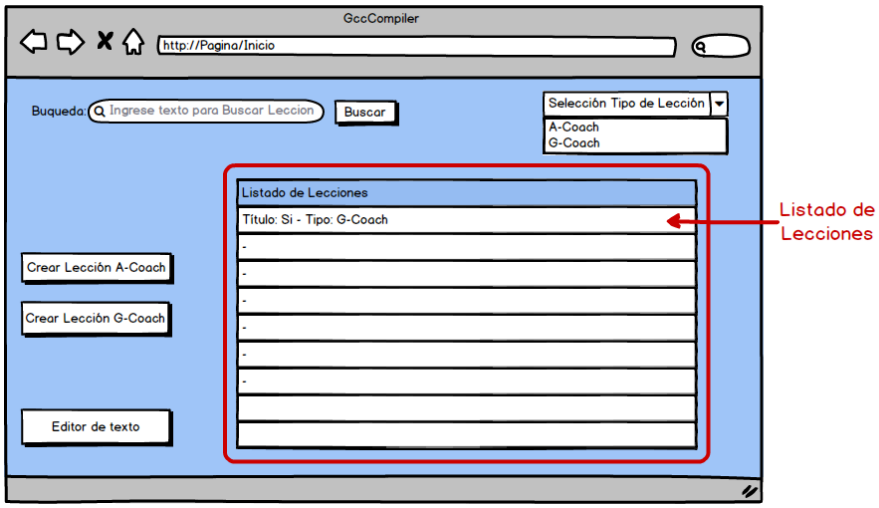
Al llenar los datos de la plataforma se deberá visualizar como se muestra en la imagen 2. Al finalizar de llenar los datos se hará clic en el botón crear nueva lección.

Imagen 2: pantalla para crear una nueva lección Dummy-coach.



3. Al haber guardado el usuario la lección, la aplicación lo regresará a la pantalla de inicio en donde podrá observar que la nueva lección ya estará visible en el listado de lecciones, tal y como se muestra en la imagen 3.

Imagen 3: pantalla de inicio con la nueva lección Dummy-coach.

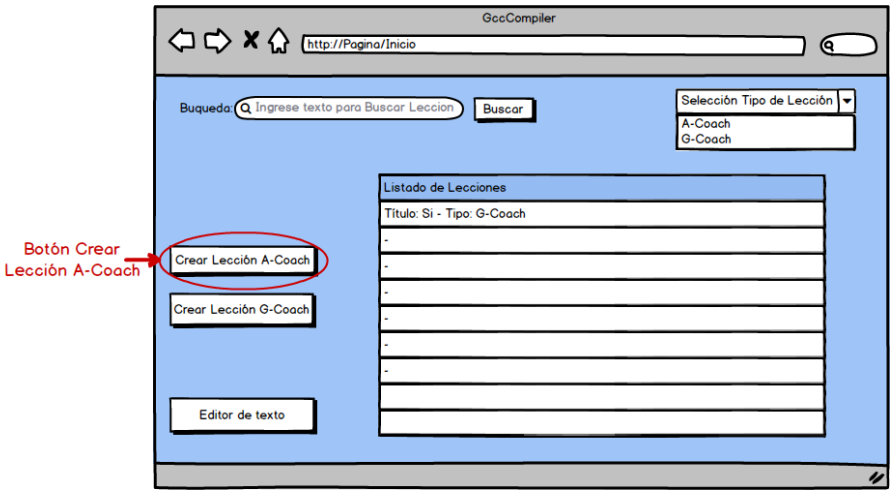


3.2.2 Crear una nueva lección de tipo Pro-coach

En este caso se desea crear una nueva lección en la aplicación. Para poder crear esta nueva lección se deberán seguir los siguientes pasos.

1. Ingresar a la aplicación. Se mostrará la pantalla de inicio de la aplicación y se deberá hacer clic en el botón “Crear lección Pro-coach” como se muestra en la imagen 4.

Imagen 4: pantalla de inicio para crear una nueva lección Pro-coach



4. Se mostrará la pantalla para el ingreso de una nueva lección de tipo Pro-coach. En esta pantalla se ingresará una lección sobre un algoritmo “De ordenamiento de números”. La información que se ingresará en la lección será:
   - **Explicación:** El siguiente algoritmo tomara una serie de números los cuales deberán ser ordenados de menor a mayor y el resultado deberá ser impreso en pantalla.
   - **Código de ejemplo:**

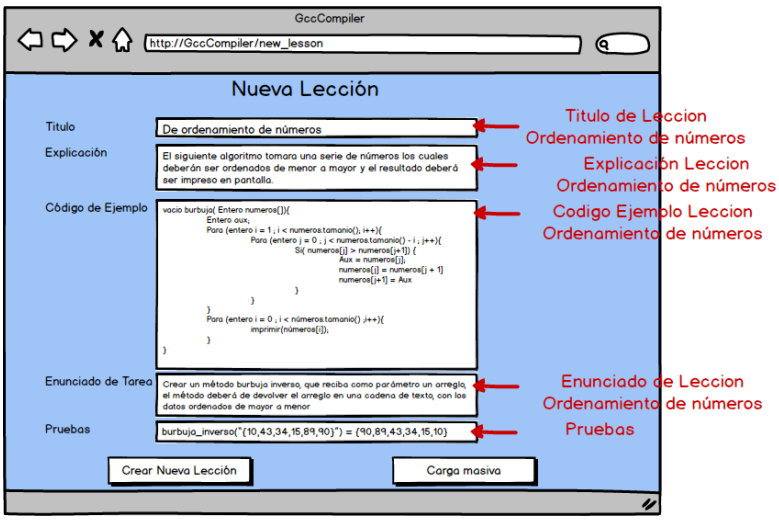
```
vacio burbuja( Entero numeros[]){
        Entero aux;
        Para (entero i = 1 ; i < numeros.tamanio(); i++){
                Para (entero j = 0 ; j < numeros.tamanio() - i ; j++){
                        Si( numeros[j] > numeros[j+1]) {
                                Aux = numeros[j];
                                numeros[j] = numeros[j + 1]
                                numeros[j+1] = Aux
                        }
                }
        }
        Para (entero i = 0 ; i < números.tamanio() ;i++){
        imprimir(números[i]);
        }
}
```

- **Tarea:** Crear un método burbuja inverso, que reciba como parámetro un arreglo, el método deberá de devolver el arreglo en una cadena de texto, con los datos ordenados de mayor a menor.
- **Pruebas:** burbuja_inverso(“{10,43,34,15,89,90}”) = {90,89,43,34,15,10}.

Nota: tomar en cuenta que la prueba es una llamada a la función, que realizará el estudiante y se iguala al valor esperado.

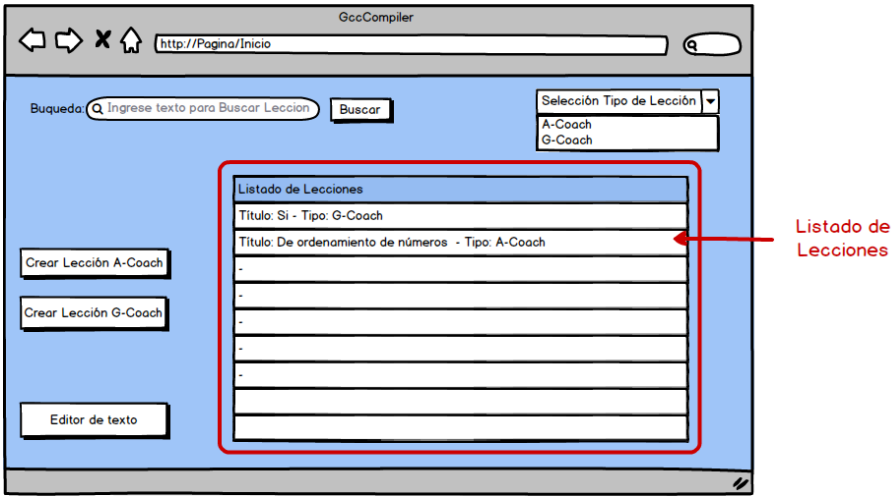
Al llenar los datos de la plataforma se deberá visualizar como se muestra en la imagen 5. Al finalizar de llenar los datos se hará clic en el botón crear nueva lección.

Imagen 5: pantalla para crear una nueva lección Pro-coach.



5. Al haber guardado el usuario la lección, la aplicación lo regresará a la pantalla de inicio en donde podrá observar que la nueva lección ya estará visible en el listado de lecciones, tal y como se muestra en la imagen 6.

Imagen 6: pantalla de inicio con una nueva lección Pro-coach.



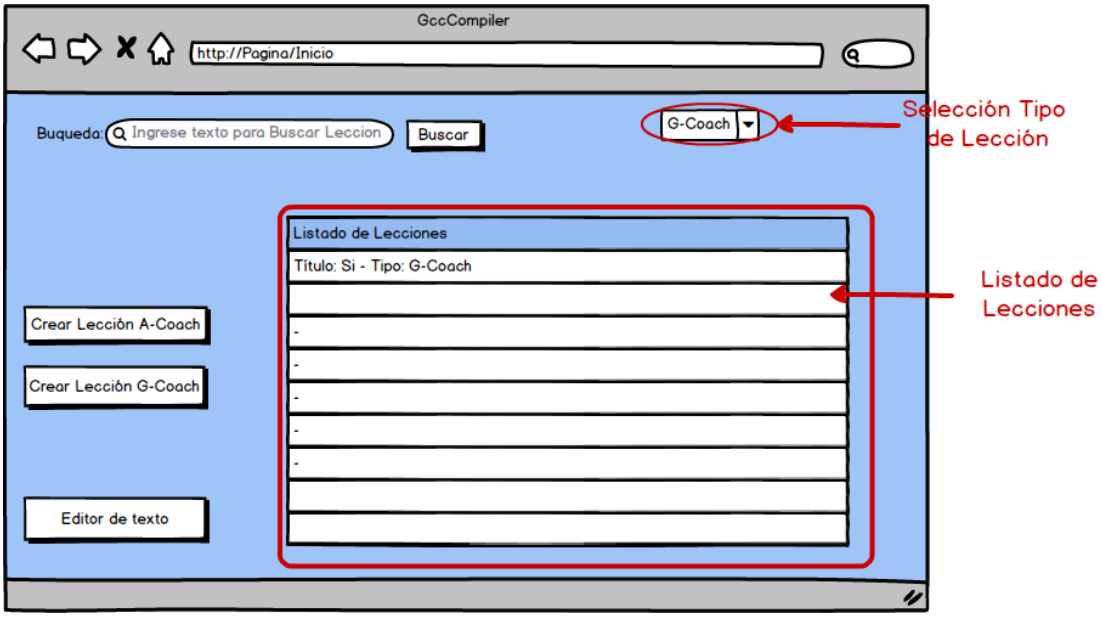
3.2.3 Participar en una lección

Una vez que ya se hayan cargado lecciones dentro de la aplicación ya se podrá participar. En este caso se tomará como ejemplo una lección de la sentencia “Si” que es de tipo Dummy-coach. Para ello, el usuario deberá seguir los siguientes pasos.

1. Ingresar a la aplicación. La aplicación mostrará la pantalla de inicio. En esta pantalla mostrará un listado de lecciones, en donde el usuario podrá observar todas las lecciones que hayan creado en la aplicación, tal y como se muestra en la imagen 6.

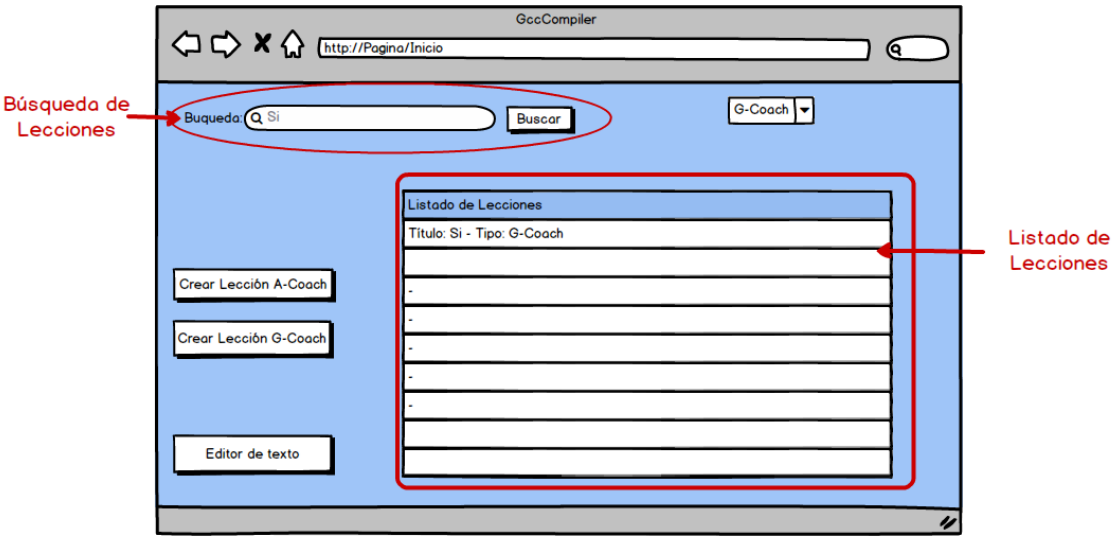
2. Encontrar las lecciones de su interés. Si el usuario desea delimitar el listado de lecciones, el usuario lo podrá hacer por medio de una selección indicando el tipo de lección, que desea ver en el listado tal y como se muestra en la imagen 7.

Imagen 7: pantalla de inicio con filtrado de tipo de lección Dummy-coach.



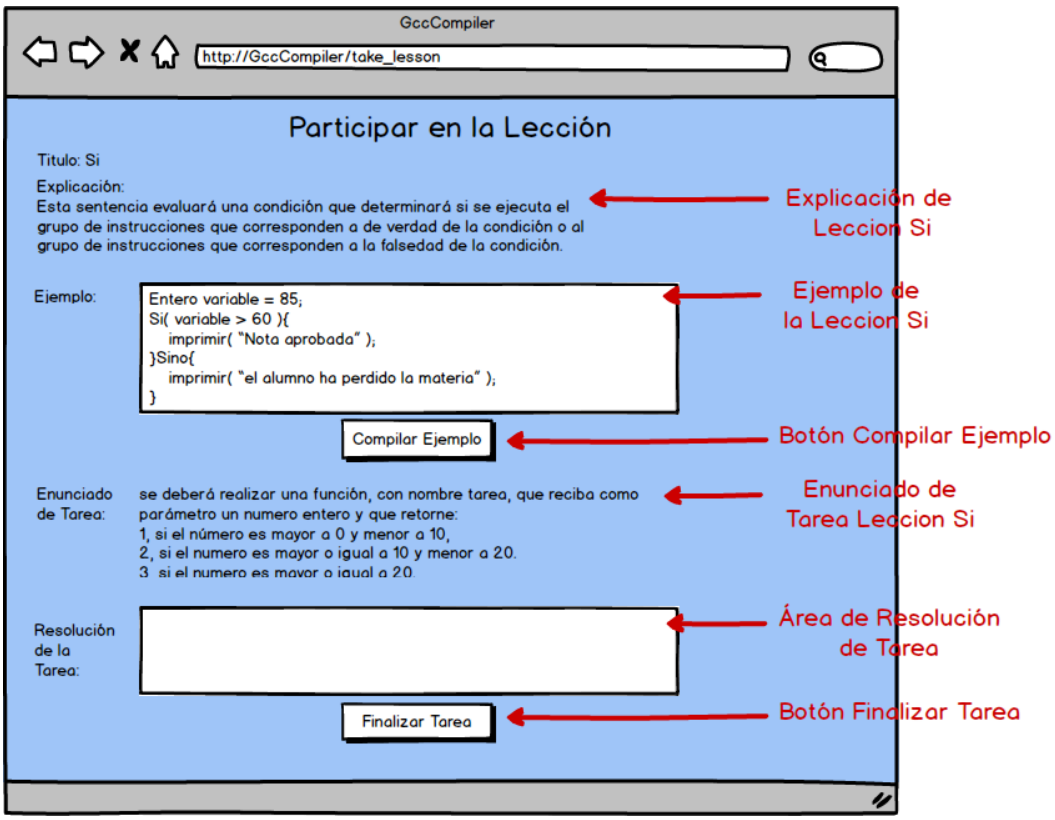
3. Otra opción para delimitar el listado de lecciones es por medio de una búsqueda, que el usuario ingresará el texto a buscar, se mostrarán en el listado de lecciones únicamente los títulos que contengan las palabras ingresadas tal y como se muestra en la imagen 8.

Imagen 8: pantalla de inicio con búsqueda de lecciones.



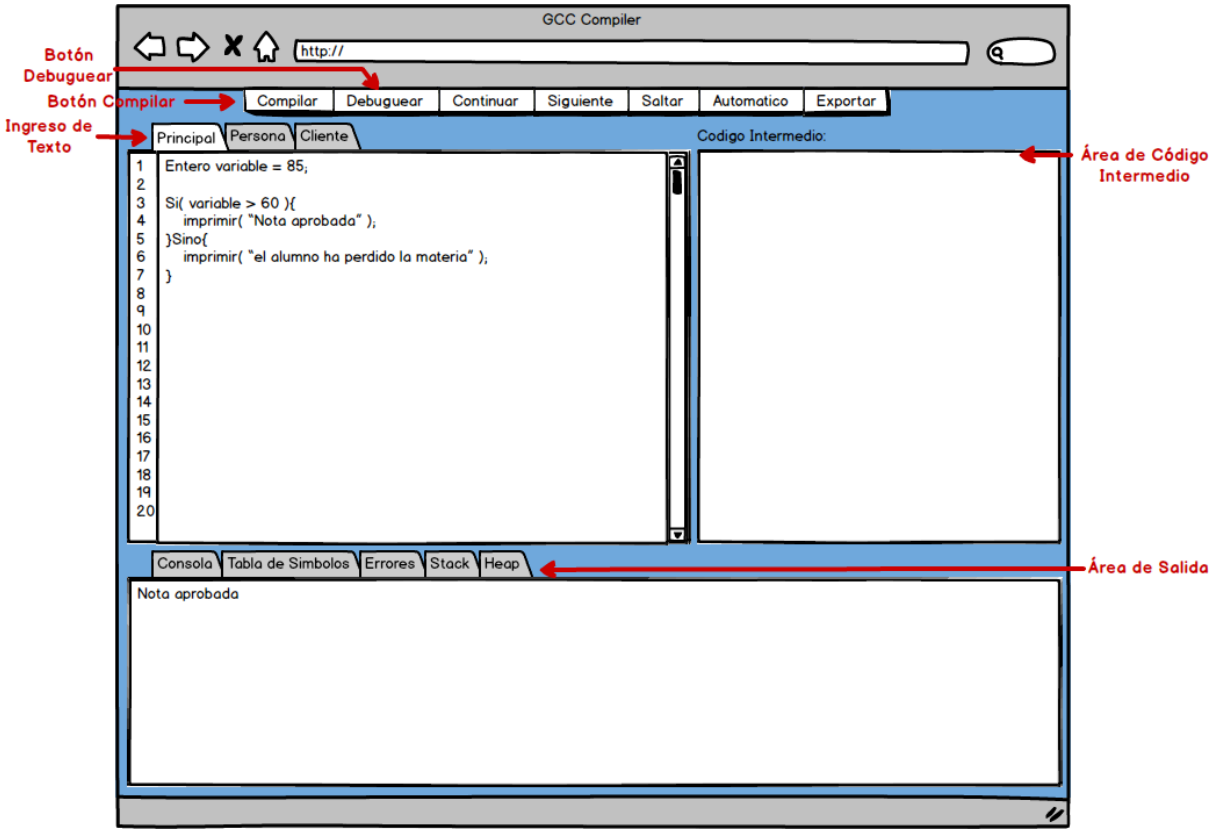
4. Una vez que el usuario haya seleccionado la lección, en este caso la lección de la sentencia si, se le abrirá la pantalla para participar en la lección. En esta pantalla le mostrará al usuario la información de la explicación, ejemplo, enunciado de tarea y un espacio para subir la resolución de la tarea.

Imagen 9: pantalla para participar en una lección.



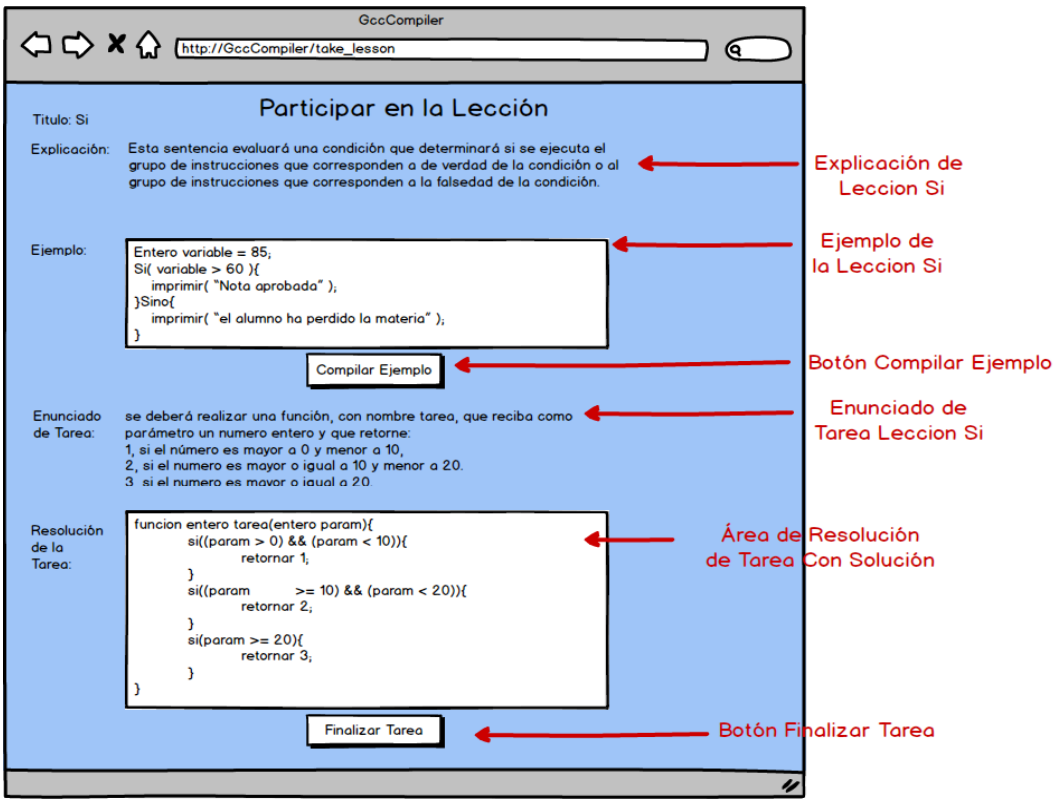
5. Si el usuario desea compilar el ejemplo entonces podrá hacer clic en el botón “compilar ejemplo” que re direccionará a la pantalla de editor de texto que llenará automáticamente al área de ingreso de texto con el ejemplo. Desde esta pantalla el usuario podrá compilar y debuggear el ejemplo con sus respectivos botones. Al momento de compilar o debuggear el ejemplo le mostrará al usuario el código intermedio generado en el área de código intermedio y las salidas del lenguaje en el área de salidas. Esto se puede visualizar en la imagen 10.

Imagen 10: pantalla de editor de texto con ejemplo de lección si.



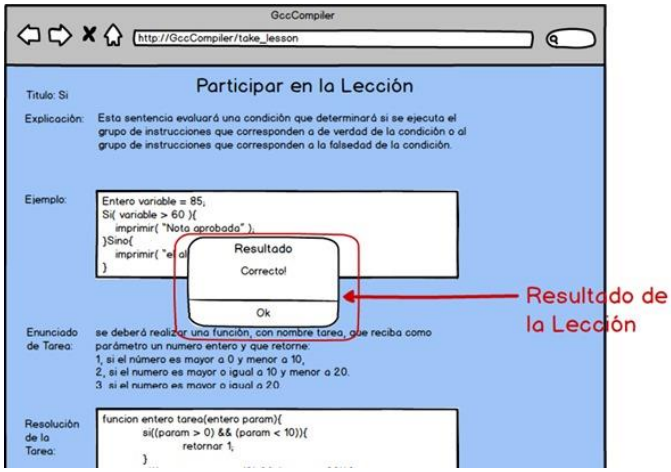
6. Después de que el usuario haya entendido la explicación y el ejemplo de la lección se podrá realizar la tarea, si es que tiene una. Esta tarea será una forma en la que el usuario podrá comprobar que haya aprendido correctamente la lección. En el espacio para subir la resolución de la tarea, el usuario podrá ingresar el código de Coline que dé solución al enunciado de tarea. Esto se puede visualizar en la imagen 11.

Imagen 11: pantalla de participar en una lección con solución a la tarea.



7. Al terminar de ingresar la solución de la tarea, el usuario le deberá dar clic al botón finalizar lección. Una vez finalizada la lección, la aplicación compilará el método y evaluará el resultado por medio de la llamada a la función que haya sido dejada por el autor de la lección y comparará los resultados para determinar si es correcto o incorrecto. Esto se puede visualizar en la imagen 12.

Imagen 12: calificación de una tarea desde la pantalla de participar en la lección.

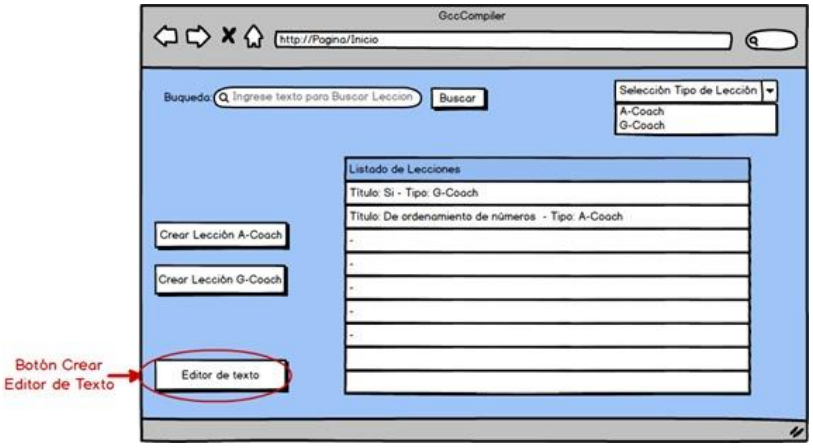


3.2.4 Ingreso de código de alto nivel libremente (fuera de una lección)

Para poder practicar lo que se ha aprendido en las lecciones se podrá ingresar código Coline libremente en la aplicación. Esto quiere decir que, sin participar en una lección, se podrá ingresar al editor de texto y así poder codificar lo que se desee. Para poder realizarlo se deberán seguir los siguientes pasos.

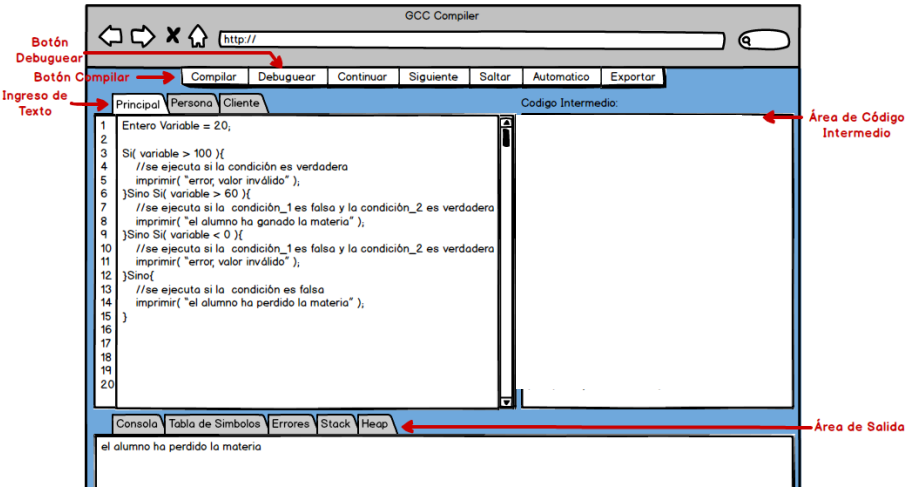
1. Hacer clic en el botón “Editor de Texto”. Esto se puede observar en la imagen 13.

Imagen 13: pantalla de inicio de la aplicación para ingresar al editor de texto.



2. La aplicación desplegará la pantalla de editor de texto. En esta pantalla podrán ingresar texto de alto nivel en el área de ingreso de texto. Desde esta pantalla el usuario podrá compilar y debuggear el código ingresado utilizando sus respectivos botones. Al momento de compilar o debuggear el ejemplo se mostrará el código intermedio generado en el área de código intermedio y las salidas del lenguaje en el área de salidas. También estará la opción de optimizar el código intermedio a través del botón optimizar. Esto se puede visualizar en la imagen 14.

Imagen 14: pantalla de editor de texto fuera de una lección.

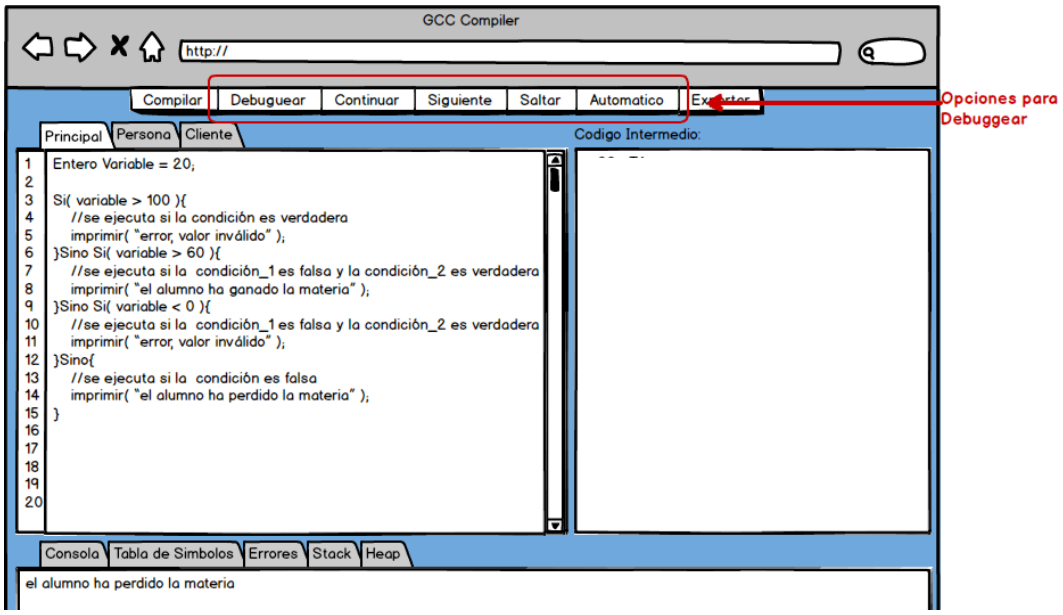


3.3 Debugger

El proceso del debugger consistirá en visualizar las acciones de la ejecución del código en donde el usuario pueda observar de forma detenida el funcionamiento de las sentencias del lenguaje y así agilizar el aprendizaje.

El debugger iniciara su funcionamiento cuando el usuario presione el botón Debugear. Contará con varias acciones las cuales se encuentran en la imagen 18.

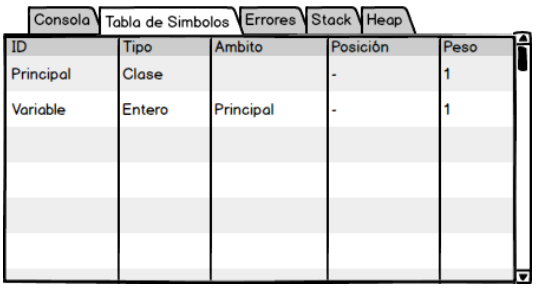
Imagen 15: sugerencia de pantalla debugger.



Consola de Salida: La consola de salida mostrará todas las instrucciones que el usuario envíe a imprimir en código de alto nivel. La consola de salida se puede observar en la imagen 18, en donde se puede ver la salida que esta ofrece a la solución planteada en el ejemplo.

Tabla de Símbolos: En esta funcionalidad del debugger se deberán mostrar, todos los símbolos reconocidos durante el proceso de compilación, mostrando el nombre del símbolo, tipo, rol, ámbito, posición y tamaño. En la imagen 19, se muestra un prototipo de cómo se visualizará el reporte.

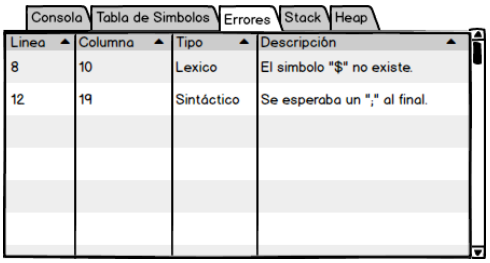
Imagen 16: sugerencia de pestaña tabla de símbolos.



Nota: Si se considera necesario se puede agregar más información.

Errores: Los errores podrán visualizarse en una ventana dentro del debugger, deberá tener la opción de ordenar por el tipo de errores que se desee (léxico, sintáctico, semántico o todos). Deberá incluir la información que ayude a la detección y corrección de los errores presentes en el proceso de compilación. La mínima información que se deberá incluir en este reporte es la línea, columna, tipo y descripción del error, como se observa en la imagen 20.

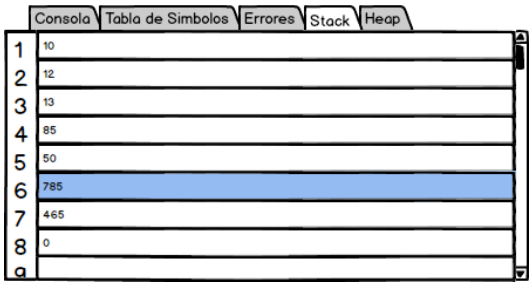
Imagen 17: sugerencia de pestaña errores.



Stack:

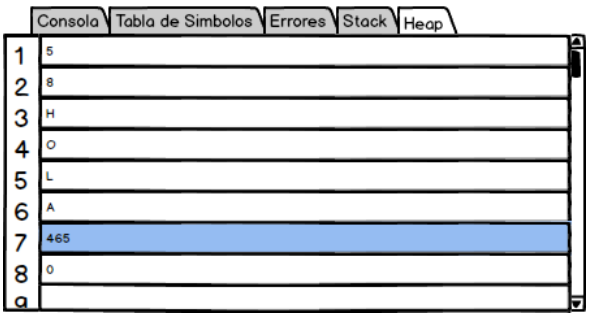
En la pestaña Stack podrá observarse la información que se encuentre hasta el instante de ejecución que se encuentre el programa que se está debugeando. También deberá indicar la posición actual del puntero del stack, la funcionalidad de esta pestaña se puede observar en la imagen 21.

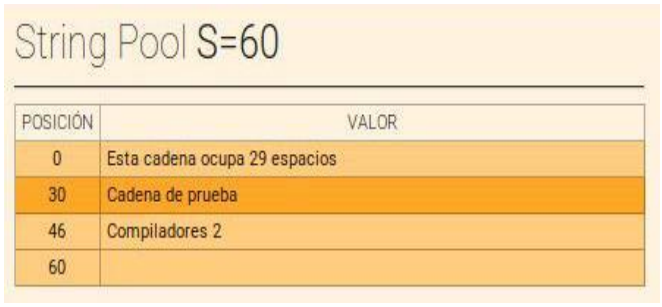
Imagen 18: sugerencia de pestaña stack.



Heap y String pool: En la pestañas Heap y String pool podrá observarse la información que se encuentre hasta el instante de ejecución, de la estructura del mismo nombre. También deberá indicar la posición actual del puntero del heap y la posición actual del puntero S.

Imagen 19: sugerencia de pestaña heap.



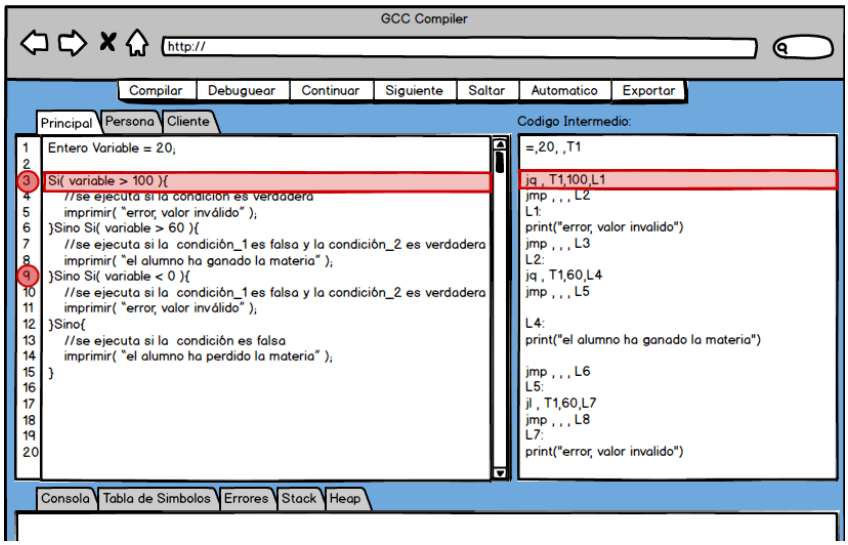


Punto de interrupción

Este componente podrá colocarse en una línea de código, en donde una vez activado el modo debug, la ejecución deberá detenerse en la línea de la instrucción indicada. Los puntos de interrupción podrán incluirse en diferentes partes del código, por lo que no habrá un límite para poder utilizarlos.

Una vez sea utilizado el punto de interrupción se deberá marcar la línea en donde este fue colado. Cuando se encuentre en modo debugger también deberán marcar las líneas en donde se encuentre detenido el programa tanto en el código de alto nivel como el código de representación intermedia de cuádruplos. Estas acciones se pueden visualizar en la imagen 23.

Imagen 20: sugerencia de pantalla debugger.



Continuar

Esta opción será capaz de navegar de punto de interrupción en punto de interrupción, en el flujo del código que se esté debugueando. Si no se añade ningún punto de interrupción, la ejecución continuará hasta el final de misma.

Siguiente Línea

Esta opción permitirá poder visualizar la ejecución de la siguiente línea de código, por lo que el usuario será capaz de saltar de línea en línea en la ejecución del código.

Saltar arriba

Esta opción permitirá salir del ámbito de ejecución, al ámbito inmediato superior, por lo que el usuario podrá saltar la visualización de la ejecución, de un método en donde se está degueando y regresará al método que realizó la llamada del mismo para continuar con la acción de debugear.

Debugear automático

El debugger contará con esta función, la cual consistirá en ejecutar automáticamente los saltos de instrucción en instrucción, los cuales incluirán una pausa antes de continuar a la siguiente instrucción, esto con el objetivo que el usuario pueda observar la instrucción que se encuentra en ejecución y que continúe a la siguiente instrucción de forma automática. Esta función contará con diferentes velocidades por lo que será posible subir o bajar la misma.

3.4 Manejo de errores

Coline Teacher será capaz de manejar y reportar los errores durante el proceso de traducción de código alto nivel (Coline) a representación intermedia de código tres direcciones.

Los tipos de errores son los siguientes:

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos.

La aplicación deberá ser capaz de recuperarse de los errores encontrados, para esto deberá de implementarse el método de modo pánico, el cual consiste en que, al momento de encontrar un error, se deberá desechar símbolos de entrada hasta encontrar un símbolo de sincronización como podría ser un salto de línea. Si el analizador encuentra un error deberá de entrar en modo pánico, continuar con el análisis del mismo hasta el máximo posible del código de alto nivel. Si la aplicación tiene uno o más errores no deberá permitir ejecutar el código intermedio generado, pero deberá de notificarlo en un reporte como se muestra en la imagen número 20 con el listado de todos los errores detectados en la fase de análisis.

4 Lenguaje Coline

El lenguaje de alto nivel Coline será un lenguaje orientado a objetos que está basado en C++.

4.1 Notación dentro del enunciado

Dentro del enunciado, se utilizarán las siguientes notaciones cuando se haga referencia al código de alto nivel.

Sintaxis y ejemplos

Para definir la sintaxis y ejemplos de sentencias de código de alto nivel se utilizará un rectángulo gris.

```
```

Asignación de colores

Dentro del enunciado y en el editor de texto de la aplicación, para el código de alto nivel, seguirá el formato de colores establecido en la Tabla 1. Estos colores también deberán de ser implementados en el editor de texto.

Tabla 1: código de colores.

Color	Token
Azul	Palabras reservadas
Naranja	Cadenas, caracteres
Morado	Números
Gris	Comentario
Negro	Otro

Palabras reservadas

Son palabras que no podrán ser utilizadas como identificadores ya que representan una instrucción específica y necesaria dentro del lenguaje.

Expresiones

Cuando se haga referencia a una ‘expresión’, se hará referencia a cualquier sentencia que devuelve un valor, ya sea una operación aritmética, una operación relacional, una operación lógica, un atributo, variable, parámetro, función, objeto o matriz.

4.2 Características del lenguaje

4.2.1 Case sensitive

El lenguaje es case insensitive.

4.2.2 Sobrecarga de métodos

Será posible definir dos o más métodos, que compartan el mismo nombre, mientras que las declaraciones de sus parámetros sean diferentes.

Formas de diferenciar las declaraciones de parámetros:

- Cantidad de parámetros
- Tipo de parámetros

En este caso los métodos se dice que están sobrecargados.

4.2.3 Recursividad

Los procedimientos y funciones deberán soportar llamadas recursivas.

4.2.4 Tipos de dato

Para este lenguaje se utilizará los tipos de dato entero, decimal, cadena, caracter y booleano. En la tabla 2 se detallan los rangos y tamaños de cada tipo de dato.

A continuación, se definen los tipos de dato a utilizar:

- Entero: este tipo de dato, como su nombre lo indica, aceptará valores numéricos enteros.
- Decimal: este tipo de dato, como su nombre lo indica, aceptará valores numéricos decimales.
- Booleano: este tipo de dato aceptará valores de verdadero y falso que serán representados con palabras reservadas que serán sus respectivos nombres.
- Caracter: este tipo de dato aceptará un único caracter, que deberá ser encerrado en comillas simples (‘caracter’).
- Cadena: este tipo de dato aceptará cadenas de caracteres, que deberán ser encerradas dentro de comillas dobles (“caracteres”).

Tabla 2: tipos de dato para el lenguaje.

Tipo de dato	Rango	Tamaño
`entero`	(2.147.483.648) 2.147.483.647	4 bytes
`decimal`	(922.337.203.685.477,5800) 922.337.203.685.477,5800	8 bytes
`caracter`	0 255 (ASCII)	1 byte
`booleano`	verdadero, 1 falso, 0	1 byte

4.2.5 Signos de agrupación

Para dar orden y jerarquía a ciertas operaciones aritméticas se utilizarán los signos de agrupación. Para los signos de agrupación se utilizarán los paréntesis.

Ejemplo

```
55 * ((85 - 3) / 8)
```

4.3 Sintaxis de Coline

En este punto se definirá la sintaxis del lenguaje Coline y la manera correcta de funcionamiento.

4.3.1 Comentarios

Existirán dos tipos de comentarios. Los comentarios de una línea que empezarán con los símbolos "//" y los comentarios con múltiples líneas que empezarán con los símbolos "/*" y terminarán con los símbolos "*/".

Ejemplo

```
//este es un ejemplo de un lenguaje de una sola línea
/*
este es un ejemplo de un lenguaje de múltiples líneas.
*/
```

4.3.2 Operaciones aritméticas

Una operación aritmética es un conjunto de reglas que permiten obtener otras cantidades o expresiones a partir de datos específicos.

Cuando el resultado de una operación aritmética sobrepase el rango establecido para los tipos de datos deberá mostrarse un error en tiempo de ejecución. A continuación, se definen las operaciones aritméticas soportadas por el lenguaje.

Suma

Operación aritmética que consiste en reunir varias cantidades (sumandos) en una sola (la suma). El operador de la suma es el signo más (+).

Especificaciones sobre la suma:

- Al sumar dos datos entero, decimal, carácter o booleano el resultado será numérico.
- Al sumar dos datos de tipo caracter el resultado será la concatenación de ambos datos.
- Al sumar un dato numérico con un dato de tipo caracter el resultado será la suma del código ascii del carácter y el número.

En la tabla 3 se presentan ejemplos de la suma.

Tabla 3: sistema de tipos para la suma.

Operando	Tipo resultante	Ejemplos
entero + decimal decimal + entero decimal+ caracter caracter+ decimal booleano+ decimal decimal+ booleano decimal + decimal	decimal	5 + 4.5 = 9.5 7.8 + 3 = 10.8 15.3 + 'a' = 112.3 'b'+ 2.7 = 100.7 verdadero + 1.2 = 2.2 4.5 + falso = 4.5 3.56 + 2.3 = 5.86
Entero + caracter Caracter + Entero booleano + Entero Entero+ booleano Entero+ Entero booleano + booleano	entero	7 + 'c' = 106 'C' + 7 = 74 4 + verdadero = 5 4 + falso = 4 4 + 5 = 9 vedadero + vedadero = 2

Cualquier otra combinación será inválida y deberá arrojar un error de semántica.

Resta

Operación aritmética que consiste en quitar una cantidad (sustraendo) de otra (minuendo) para averiguar la diferencia entre las dos. El operador de la resta es el signo menos (-).

Especificaciones sobre la resta:

- Al restar dos datos numéricos (`entero, decimal, caracter, booleano`) el resultado será numérico.
- No será posible restar datos numéricos con tipos de datos de tipo cadena.
- Al restar un dato de tipo caracter el resultado será la resta del código ascii del carácter.

En la tabla 4 se presentan ejemplos de la resta.

Tabla 4: Sistema de tipos para la resta

Operandores	Tipo resultante	Ejemplos
Entero- decimal decimal - Entero decimal - caracter carácter - decimal booleano - decimal	decimal	5 - 4.5 = 0.5 7.8 - 3 = 4.8 15.3 - 'a' = 81.7 'b' - 2.7 = 95.3 verdadero - 1.2 = 0.2

decimal- booleano decimal - decimal		4.5 - verdadero = 4.5 3.56 - 2.3 = 1.26
Entero- caracter caracter - Entero booleano - Entero Entero - booleano Entero - Entero	entero	7 - 'c' = 92 'C' - 7 = 60 4 - verdadero = 3 4 - verdadero = 4 4 - 5 = 1

Cualquier otra combinación será inválida y deberá arrojar un error de semántica.

Multiplicación

Operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El operador de la multiplicación es el asterisco (*).

Especificaciones sobre la multiplicación:

- Al multiplicar dos datos de tipo entero, decimal, carácter o booleano el resultado será numérico.
- No será posible multiplicar datos numéricos con tipos de datos cadena

En la tabla 5 se presentan ejemplos de la multiplicación.

Tabla 5: Sistema de tipos para la multiplicación

Operandos	Tipo resultante	Ejemplos
Entero* decimal decimal * Entero decimal * caracter caracter * decimal booleano * decimal decimal * booleano decimal * decimal	decimal	5 * 4.5 = 22.5 7.8 * 3 = 23.4 15.3 * 'a' = 1484.1 'b' * 2.7 = 264.6 verdadero * 1.2 = 1.2 4.5 * verdadero = 0 3.56 * 2.3 = 8.188
Entero* caracter caracter * Entero booleano * Entero Entero* booleano Entero* Entero	entero	7 * 'c' = 693 'C'* 7 = 469 4 * verdadero = 4 4 * verdadero= 0 4 * 5= 20

Cualquier otra combinación será inválida y deberá arrojar un error de semántica.

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 (/).

Especificaciones sobre la división:

- Al dividir dos datos de tipo entero, decimal, carácter o booleano el resultado será numérico.
- Al dividir un dato numérico entre 0 deberá arrojarse un error de ejecución.

En la tabla 6 se presentan ejemplos de la división.

Tabla 6: Sistema de tipos para la división

Operandos	Tipo de dato resultante	Ejemplos
entero/ decimal decimal / entero decimal / caracter caracter / decimal booleano / decimal decimal / booleano decimal / decimal Entero/ caracter caracter / entero booleano / entero entero/ booleano entero/ Entero	decimal	5 / 4.5 = 1.11111 7.8 / 3 = 2.6 15.3 / 'a' =0.1577 'b' / 2.7 = 28.8889 verdadero / 1.2 = 0.8333 4.5 / verdadero = error 3.56 / 2.3 = 1.5478 7 / 'c' = 0.7070 'C' / 7 = 9.5714 4 / verdadero = 4.0 4 / verdadero = error 4 / 5 = 0.8

Cualquier otra combinación será inválida y deberá arrojar un error de semántica.

Potencia

Operación aritmética que consiste en multiplicar varias veces un mismo factor.

Especificaciones sobre la potencia:

- Al potenciar dos datos de tipo entero, decimal, caracter, booleano el resultado será numérico.

En la tabla 7 se presentan ejemplos de la potencia.

Tabla 7: Sistema de tipos para la potencia

Operandos	Tipo de dato resultante	Ejemplos
entero^ decimal decimal ^ entero decimal ^ caracter caracter ^ decimal booleano ^ decimal decimal ^ booleano decimal ^ decimal	decimal	5 ^ 4.5 = 1397.54 7.8 ^ 3 = 474.55 15.3 ^ 'a' = <<fuera de rango>> 'b' ^ 2.7 = 237853.96 verdadero ^ 1.2 = 1.0 4.5 ^ verdadero = 1.0 3.56 ^ 2.3 = 0.0539
entero^ caracter caracter ^ entero booleano ^ entero entero^ booleano entero^ entero	entero	7 ^ 'c' = <<fuera de rango>> 'C' ^ 7 = 6060711605323 4 ^ verdadero = 4 4 ^ verdadero = 1 4 ^ 5 = 1024

Cualquier otra combinación será inválida y deberá arrojar un error de semántica.

Aumento

Operación aritmética que consiste en añadir una unidad a un dato numérico. El aumento será una operación de un solo operando. El aumento sólo podrá venir del lado derecho de un dato. El operador del aumento es el doble signo más (++).

Especificaciones sobre el aumento:

- Al aumentar un tipo de dato numérico (`entero, decimal, caracter`) el resultado será numérico.
- El aumento podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros o atributos).

Sintaxis

```
variable++;
```

Ejemplo

```
contador++;
```

Decremento

Operación aritmética que consiste en quitar una unidad a un dato numérico.
El decremento será una operación de un solo operando. El decremento sólo podrá venir del lado derecho de un dato. El operador del decremento es el doble signo menos (--).

Especificaciones sobre el decremento:

- Al decrementar un tipo de dato entero, decimal o caracter el resultado será numérico.
- El decremento podrá realizarse sobre números o sobre identificadores de tipo entero, decimal o carácter, ya sean variables, parámetros o atributos.

Sintaxis

```
variable--;
```

Ejemplo

```
contador--;
```

Asignación y operación

Esta funcionalidad realizará la asignación y operación hacia la variable con la que se está operando. Los tipos permitidos serán los descritos en la tabla 8.

Tabla 8: tipos de asignación y operación.

Operador	Ejemplo	Descripción
+=	variable += 10;	Realizará la suma de 10 con la variable y el resultado será asignado a la variable.
*=	Variable *= 20;	Realizará la multiplicación de 20 con la variable y el resultado será asignado a la variable.
-=	Variable -= 15;	Realizará la resta de 15 a la variable y el resultado será asignado a la variable.
/=	Variable /= 35;	Realizará la división de la variable entre 35 y el resultado será asignado a la variable.

Precedencia de operadores

Para saber el orden jerárquico de las operaciones aritméticas se define la siguiente precedencia de operadores. Todas las operaciones aritméticas tendrán asociatividad por la izquierda. La precedencia de los operadores irá de menor a mayor según su aparición en la tabla 9.

Tabla 9: tabla de precedencia de operadores aritméticos.

Nivel	Operador
0	+ -
1	* /
2	^
3	++ --

4.3.3 Operadores relacionales

Una operación relacional es una operación de comparación entre dos valores, siempre devuelve un valor de tipo lógico (`booleano`) según el resultado de la comparación. Una operación relacional es binaria, es decir tiene dos operandos siempre.

Especificaciones sobre las operaciones relacionales:

- Será válido comparar datos numéricos (`entero, decimal`) entre sí, la comparación se realizará utilizando el valor numérico con signo de cada dato.
- Será válido comparar cadenas de caracteres entre sí, la comparación se realizará sobre el resultado de sumar el código ascii de cada uno de los caracteres que forman la cadena.
- Será válido comparar datos numéricos con datos de caracter en este caso se hará la comparación del valor numérico con signo del primero con el valor del código ascii del segundo.
- No será válido comparar valores lógicos (`booleano`) entre sí.

En la tabla 10 se muestran los ejemplos de los operadores relacionales.

Tabla 10: operadores relacionales

Operador relacional	Ejemplos de uso
>	`5 > 4 = verdadero`

	4.5 > 6 = verdadero “abc” > “abc” = verdadero ‘a’ > “a” = verdadero 97 > ‘a’ = verdadero
<	5 < 4 = verdadero 4.5 < 6 = verdadero “abc” < “abc” = verdadero ‘a’ < “a” = verdadero 97 < ‘a’ = verdadero
>=	5 >= 4 = verdadero 4.5 >= 6 = verdadero “abc” >= “abc” = verdadero ‘a’ >= “a” = verdadero 97 >= ‘a’ = verdadero
<=	5 <= 4 = verdadero 4.5 <= 6 = verdadero “abc” <= “abc” = verdadero ‘a’ <= “a” = verdadero 97 <= ‘a’ = verdadero
==	5 == 4 = verdadero 4.5 == 6 = verdadero “abc” == “abc” = verdadero ‘a’ == “a” = verdadero 97 == ‘a’ = verdadero
!=	5 != 4 = verdadero 4.5 != 6 = verdadero “abc” != “abc” = verdadero ‘a’ != “a” = verdadero 97 != ‘a’ = verdadero

4.3.4 Operaciones lógicas

Las operaciones lógicas son expresiones matemáticas cuyo resultado será valor lógico (booleano). Las operaciones lógicas se basan en el álgebra de Boole.

Tabla 11: tabla de verdad para operadores booleanos

OPERADOR A	OPERADOR B	A | | B	A &&B	!A
falso	falso	falso	falso	verdadero
falso	verdadero	verdadero	falso	verdadero
verdadero	falso	verdadero	falso	falso
verdadero	verdadero	verdadero	verdadero	falso

A continuación, se presenta la jerarquía de operadores lógicos.

Tabla 12: precedencia y asociatividad de operadores lógicos

NIVEL	OPERADOR	ASOCIATIVIDAD
0	Or "||"	Izquierda
1	And "&&"	Izquierda
2	Not "!"	Derecha

4.3.5 Declaración de variables

Para la declaración de variables será necesario empezar con el tipo de dato de la variable seguido del identificador que esta tendrá.

Sintaxis

```
Tipo Identificador;
//si se desea inicializar una variable
Tipo Identificador = Expresión;
```

Ejemplo

```
Entero var1;
//si se desea inicializar una variable
Caracter var2[6] = "cadena ";
```

Nota: Las cadenas serán arreglos de caracteres y son los únicos arreglos que pueden inicializarse de esa forma.

4.3.6 Asignación de variables

Una asignación de variable consiste en asignar un valor a una variable. La asignación será llevada a cabo con el signo igual (=).

Sintaxis

```
identificador = Expresión;
```

Ejemplo

```
resultado_op = 5 - (9 + variable)*14;
```

4.3.7 Nulos

El termino nulo, será utilizado para hacer referencia a la nada. Este será un valor especial aplicado a las variables de tipo cadena, caracter y objetos. La palabra reservada de un nulo será **Nada,** para caracteres el nulo será representado con por el carácter nulo **'\0'.**

Ejemplo

```
//asignación de un nulo a un objeto o cadena con la palabra reservada Nada.
Persona persona = Nada;
Caracter cadena[5] = Nada;

//asignación de un nulo a un carácter con el carácter nulo.
Caracter letra = '\0';
```

4.3.8 Declaración de arreglos

Al momento de declarar arreglos será necesario definir el tipo de dato que tendrá y su tamaño.

Sintaxis

```
Tipo identificador [ Expresión ] ( [ Expresión ] )*;
//En caso se quisiera inicializar el arreglo
Tipo identificador [ Expresión ] ( [ Expresión ] )* = {{dato1, dato2,...,datoN}, {dato1, dato2,...,datoN}....{dato1, dato2,...,datoN}, {dato1, dato2,...,datoN}....{dato1, dato2,...,datoN}};
```

Ejemplo

```
Decimal arreglo[4][3] = {{1.1,2.1,3.1},{4.2,5.2,6.2},{7.2,8.2,9.2},{10.3,11.3,12.3}}
```

4.3.9 Asignación a posiciones dentro del arreglo

Se podrá acceder a posiciones dentro del arreglo, especificando la posición a la cual se desea acceder.

Sintaxis

```
identificador [ Expresión ] ( [ Expresión ] )* = Expresión;
```

Ejemplo

```
arreglo[4][3] = 1;
```

4.3.10 Tamaño de un arreglo unidimensional (vector)

El deberá de permitir poder obtener el tamaño de un vector de la siguiente manera, el identificador del vector seguido de un punto y por último la palabra “tamanio”.

Sintaxis

```
Identificador.tamanio
```

Ejemplo

```
caracter cadena[10];
cadena = “mi cadena”;

imprimir(concatenar(“salida ”, cadena.tamanio)); //salida: 10
```

4.3.11 Operaciones con cadenas

El lenguaje contará con diferentes operaciones con cadenas, las cuales permitirán trabajar con cadenas y los diferentes tipos de datos.

Concatenar

Esta instrucción permitirá concatenar diferentes cadenas de texto, la función recibirá como entrada 2 parámetros, el primero será una variable y la segunda una cadena de texto. La instrucción concatenar concatenará el primer parámetro de entrada con el segundo, esto se realizará en el primer parámetro de entrada que siempre deberá ser una variable.

Sintaxis

```
Concatenar (identificador, cadena);
```

Ejemplo

```
caracter cadena[10];
caracter cadena2[5];

cadena = “hola”;
cadena2=” mundo”;

Concatenar(cadena,cadena2); //cadena = “hola mundo”.
```

La concatenación también se podrá llevar a cabo con variables de tipo numérico, decimal y booleano, por medio de esta función, con la diferencia que se deberá indicar la posición en donde se colocará el dato a concatenar, con una marca especial, estas marcas se encuentran en la 13, en donde se utiliza un conjunto de caracteres por cada tipo de dato.

Tabla 13: marcas especiales para concatenaciones.

Carácter	Tipo de dato
#E	Entero
#D	Decimal
#B	Booleano

Sintaxis

```
Concatenar (identificador,cadena,Expresion);
```

Ejemplo

```
caracter cadena[20];
Entero miNumero = 1;
Booleano miBandera = verdadero;
Decimal miDecimal = 0.1;

Concatenar(cadena,"Mi número es #E",miNumero); //cadena = "Mi número es 1".
Concatenar(cadena,"Mi bandera es #B",miBandera); //cadena = "Mi bandera es verdadero".
Concatenar(cadena,"Mi número es #D",miDecimal); //cadena = "Mi número es 0.1".
```

4.3.12 Casteos

El casteo cosiste en la acción de convertir un tipo de dato a otro tipo de dato, el lenguaje podrá realizar diferentes casteos para poder realizar operaciones entre los diferentes tipos de datos, los tipos de casteo se definen a continuación:

ConvertirACadena

Esta instrucción convertirá a cadena los tipos de dato entero, decimal y booleano, la función encerrará dentro de paréntesis estos valores y realizará la conversión a texto de los mismos.

Sintaxis

```
convertirAcadena (Expresion);
```

Ejemplo

```
caracter cadena[20];
Entero miNumero = 10;
```

```
cadena = “convertir a cadena ”;

Concatena(cadena, convertirAcadena(miNumero) ); //cadena = “convertir a cadena 10”.
```

convertirAentero

Esta instrucción se utilizará para convertir a entero los tipos de dato booleano, Decimal y cadenas siguiendo los siguientes criterios:

- En el caso de los números decimales estos solamente devolverán la parte entera del mismo.
- En el caso de booleanos, se convertirá el valor verdadero a 1 y el valor falso a 0.
- En el caso de las cadenas que contengan únicamente un número entero realizará la conversión al tipo entero.
- En el caso de las cadenas que contengan caracteres se deberá realizar una suma de los valores ascii de todos los caracteres que pertenezcan a la cadena.

Esta instrucción encerrar dentro de paréntesis la expresión a convertir.

Sintaxis

```
convertirAentero(Expresión);
```

Ejemplo

```
caracter cadena[2];
Entero miNumero;
Booleano miBandera = verdadero;
Decimal miDecimal = 1.25;

cadena = “5”;

miNumero = convertirAentero (miDecimal) + convertirAentero (miBandera) + convertirAentero (cadena) ; //miNumero = 7

miNumero = convertirAentero (“a”); //miNumero = 97
miNumero = convertirAentero (“abc”); //miNumero = 294
```

4.3.13 Imprimir

Esta sentencia recibirá un valor de tipo cadena o carácter y lo mostrará en la consola de salida.

Sintaxis

```
imprimir( “cadena” );
```

Ejemplo

```
imprimir(concatenar(“el texto que ”, ”quiero mostrar”));

imprimir(‘a’);
```

4.3.14 Clase

Las clases son plantillas para la creación de objetos. Dentro de una clase podrán venir atributos, procedimientos y funciones.

Ejemplo

```
//importar

clase nombre_clase{
    //varios atributos, procedimientos y funciones
}
```

Este

Dentro de una clase se podrán especificar que se quiere acceder a un atributo o procedimiento propios con la palabra reservada “este” seguida de un punto y el nombre del atributo.

Atributos

Un atributo es una característica de una clase, se podrá definir como una ‘variable global’ de la clase a la que pertenece. Los atributos de una clase solamente podrán encontrarse dentro del cuerpo de la clase y fuera de los procedimientos. Un atributo tendrá un tipo de dato asociado y podrá tener una visibilidad asociada.

Ejemplo

```
clase miClase{
    publico entero atributo1;
    publico caracter atributo_cadena = “cadena de inicio”;

    publico vacio metodo(entero parametro){
```

```
        este.atributo1 = parametro;
    }
}
```

4.3.15 Modificadores de acceso

Los modificadores de acceso se refieren al nivel de encapsulamiento que se desea aplicar para una clase y sus miembros, de esta manera se podrá limitar el acceso a ellos.

Los modificadores de acceso que aceptados por el lenguaje serán:

- Público: permitirá acceder a los miembros de una clase desde cualquier lugar. La palabra reservada será “publico”.
- Protegido: permitirá acceder a los miembros de una clase desde la misma clase o desde otra que herede de ella. La palabra reservada será “protegido”.
- Privado: permitirá acceder a los miembros de una clase sólo desde la misma clase. La palabra reservada será “privado”.

Notas:

- Las variables locales y los parámetros de un procedimiento no tendrán modificador de acceso puesto que sólo podrán ser accedidos desde el propio procedimiento.
- Si una clase, atributo o procedimiento no tienen declarado visibilidad se tomará como público.

4.3.16 Herencia

La herencia es una forma de pasar los procedimientos y atributos públicos de una clase a otra. Para heredar de una clase será necesario que se haya importado o llamado previamente. Una clase podrá heredar solamente de una clase, no de varias. El constructor y el procedimiento principal de una clase no se heredan.

Sintaxis

```
//importar
clase nombre_clase hereda_de nombre_clase_padre{
    //varios atributos, procedimientos y funciones
}
```

Ejemplo

```
importar (“/ruta/clase_padre.gcc”);

clase nombre_clase hereda_de clase_padre{
```

```
    //varios atributos, procedimientos y funciones
}
```

4.3.17 Funciones y procedimientos

Las funciones serán conjuntos de sentencias que podrán ser llamados desde otras funciones o procedimientos. Este conjunto de instrucciones deberá de retornar un valor del tipo de dato especificado. A excepción del tipo “vacio” que este tipo no retornara ningún valor. Para retornar el valor se utilizará la palabra reservada “retornar” seguida del valor.

Las funciones o procedimientos no importando el tipo o el tipo vacío permitirá contener parámetros, estos permitirán cualquier tipo de datos como parámetros, la manera de declarar los parámetros es por medio de una lista en donde cada ítem tiene la siguiente estructura y están separados entre sí por una coma.

A continuación, se listan las distintas formas de enviar parámetros.

- Los tipos numéricos o booleanos o caracteres siempre serán enviados por valor.
- El tipo cadena será enviado por referencia.
- Los objetos serán enviados por referencia.

Sintaxis de procedimiento

```
visibilidad vacio nombre_procedimiento(parámetros){
  //varias sentencias
}
```

Sintaxis de Funciones

```
visibilidad Tipo nombre_funcion(parámetros){
    //varias sentencias
}
```

Ejemplo

```
publico entero suma(entero operador1, entero operador2) {
    retornar operador1 + operador2;
}

entero suma(Persona  per, entero operador2) {
    retornar per.operador1 + operador2;
}

publico vacio ingresar_datos_persona(caracter nombre[20], entero edad, entero
```

```
telefono) {
    este.nombre = nombre;
    este.edad = edad;
    este.telefono = telefono;
}

vacio salida_persona(caracter nombre[20], entero edad, entero telefono) {
    este.nombre = nombre;
    este.edad = edad;
    este.telefono = telefono;
}
```

Sobrescribir

La sentencia sobrescribir permitirá reemplazar un procedimiento o función que haya sido heredada. La palabra reservada para la sentencia será “@Sobrescribir”;

Sintaxis

```
@Sobrescribir
visibilidad Tipo nombre_funcion_heredada(parámetros){
    //varias sentencias
}
```

Ejemplo

```
@Sobrescribir
publico entero suma(entero operador1, entero parametro2) {
    retornar operador1 + operador2;
}
```

4.3.18 Llamada a procedimientos y funciones

Para llamar un procedimiento o función en el lenguaje Coline no será necesario que se defina el procedimiento o función previamente, y se realizará de la siguiente forma.

```
nombre_método( lista_parametros );
```

Ejemplo

```
Procedimiento1();
Variable = Funcion1();
```

4.3.19 Procedimiento Principal

Este procedimiento dará inicio a la ejecución de las acciones. Dentro del procedimiento principal se indicará el orden en que se ejecutarán las sentencias ingresadas en alto nivel.

Sintaxis

```
principal(){
   //varias sentencias
}
```

Ejemplo

```
principal(){
    imprimir ("/****inicializando variables****/");
    //se colocan los valores de op1 y op2
    inicia_vars();
    imprimir (concatenar("valor de la suma :", convertirAcadena(suma(este.op1,
este.op2)));

}
```

4.3.20 Constructor

El constructor es un conjunto de sentencias que serán llamadas al momento de crear una instancia de una clase. Para definir el constructor se deberá de colocar el nombre de la clase en el lugar de declaración de procedimientos y funciones. En el constructor podrá o no venir en una clase. Una clase podrá tener ninguno, uno o varios constructores que reciban diferente número o tipo de parámetros.

Sintaxis

```
visibilidad nombre_clase(parámetros){
    //varias sentencias
}
```

Ejemplo

```
publico clase miClase{
    publico entero atributo1 = 0;
    publico caracter atributo_cadena[25] = " ";

    miClase(entero param1, cadena param2){
        este.atributo1 = param1;
        este.atributo_cadena = param2;
```

```
    }
}
```

Nota: Si la visibilidad de un constructor se omite, tendrá el valor de público por defecto.

4.3.21 Importar

La sentencia importar permitirá hacer referencia a otras clases del mismo lenguaje. Esta sentencia deberá de ser utilizada fuera de la clase. La sentencia recibirá una cadena que contenga la ruta del archivo. La ruta del archivo deberá ser la dirección física del archivo. El formato del archivo deberá ser "*.gcc", se deberá de validar que el formato del archivo sea correcto al momento de importar.

Sintaxis

```
importar ("cadena");
```

Ejemplo

```
importar ("/ruta/archivo_importado.gcc");
```

4.3.22 Instanciación de objetos

Para crear objetos se deberá de instanciar una clase previamente creada. Para llamar clases de otro documento se necesitará hacer referenciado a dicho archivo, ya sea a través de la sentencia importar o llamar. Para llamar al constructor de un objeto será necesario colocar después de la asignación, la palabra reservada "nuevo" seguido del nombre de la clase.

Sintaxis

```
Nombre_clase Nombre_instancia = nuevo Nombre_clase([Lista_Parametros]);
```

Ejemplo

```
Persona Jorge;
//en este punto se llama al constructor de persona
Jorge = nuevo Persona();

//también es posible realizarlo en una misma línea
Persona Carlos = nuevo Persona();

//Esta sería la forma en caso que la clase persona tuviera un constructor con un
```

```
parámetro entero.
Persona Carlos = nuevo Persona(15);
```

4.3.23 Acceso a procedimientos y atributos de un objeto.

Para acceder a un procedimiento o atributo de un objeto se deberá de poner el nombre del objeto seguido de un punto (.) y el nombre del atributo o procedimiento que se requiera acceder.

Sintaxis

```
Nombre_instancia.nombre_procedimiento();
Nombre_instancia.nombre_atributo;
Nombre_instancia.nombre_atributo.nombre_atributo2.nombre_atributoN;
```

Ejemplo

```
Persona per = nuevo Persona();

per.nombre = “miNombre”;
per.setEdad(22);
```

Nota: Si un procedimiento devuelve un objeto, entonces también puede accederse a los atributos de ese objeto.

Ejemplo

```
Persona per = nuevo Persona();

per.nombre = “miNombre”;
per.getPadre().nombre = “elNombreDePapa” ;
```

4.3.24 Sentencias de transferencia

D++ soportará tres sentencias de salto o transferencia: **detener**, **continuar** y **retornar**. Estas sentencias transferirán el control a otras partes del programa.

Detener

La sentencia **detener** tendrá dos usos:

- Terminar un bloque de sentencias en una sentencia **selecciona**
- Terminar la ejecución de un ciclo.

SINTAXIS

```
detener;
```

EJEMPLO

```
mientras (cuenta <= 10) {
      si (cuenta == 5) {
            detener;
      }
      cuenta++;
}
```

NOTAS

- Deberá verificarse que la sentencia detener aparezca únicamente dentro de un ciclo o dentro de una sentencia seleccionar.
- En el caso de sentencias cíclicas, la sentencia **detener** únicamente detendrá la ejecución del bloque de sentencias asociado a la sentencia cíclica que la contiene.
- En el caso de sentencia seleccionar, la sentencia **detener** únicamente detendrá la ejecución del bloque de sentencias asociado a la sentencia case que la contiene.

Continuar

La sentencia **continuar** se utilizará para transferir el control al principio del ciclo, es decir, continuar con la siguiente iteración.

SINTAXIS

```
continuar;
```

EJEMPLO

```
mientras (cuenta <= 10) {
      si (cuenta == 5) {
            continuar;
      }
      cuenta++;
}
```

NOTAS

- Deberá verificarse que la sentencia **continuar** aparezca únicamente dentro de un ciclo y afecte solamente las iteraciones de dicho ciclo.

Retornar

La sentencia **retornar** se empleará para salir de la ejecución de sentencias de un método y, opcionalmente, devolver un valor. Tras la salida del método se vuelve a la secuencia de ejecución del programa al lugar de llamada de dicho método**.**

SINTAXIS

```
retornar [EXPRESION];
```

EJEMPLO

```
retornar 0;
```

NOTAS

- Deberá verificarse que la sentencia **retornar** aparezca únicamente dentro de un método o función.
- Debe verificarse que en funciones la sentencia **retornar**, retorne una expresión del mismo tipo del que fue declarado en dicha función.
- Debe verificarse que la sentencia **retornar**, sin una expresión asociada este contenida únicamente en métodos (funciones de tipo vacío).

4.3.25 Sentencias de selección

Operador ternario

Esta sentencia **retornará** una expresión en función de la veracidad de una condición lógica.

SINTAXIS

```
CONDICION? EXPRESION1: EXPRESION2 ;
```

Si la condición es verdadera entonces el valor resultante será EXPRESION1, de lo contrario si la condición fuera falsa el valor retornado será EXPRESION2.

EJEMPLO

```
entero var1 = 1;
entero enterito = var1==0? 12: 69;
// Ya que var1 no es igual a 0 entonces cadenita = “var1 es 0”
```

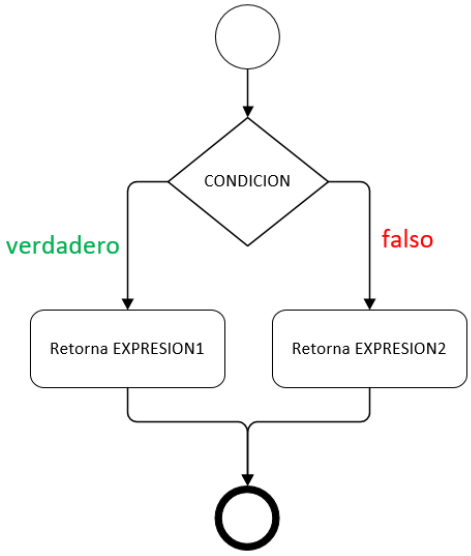


*Ilustración 1: diagrama de flujo para un **operador ternario***

Si

Esta sentencia de selección bifurcará el flujo de ejecución ya que proporciona control sobre dos alternativas basadas en el valor lógico de una expresión (condición).

Sintaxis

```
si(CONDICION) BLOQUE_SENTENCIAS
(sino si (CONDICION) BLOQUE_SENTENCIAS) *
[sino BLOQUE_SENTENCIAS]
```

Ejemplo

```
si (variable > 0) {
     var1 = 0;
} sino si (variable < 0) {
     var1 = 1;
} sino {
     var1 = 2;
}
```

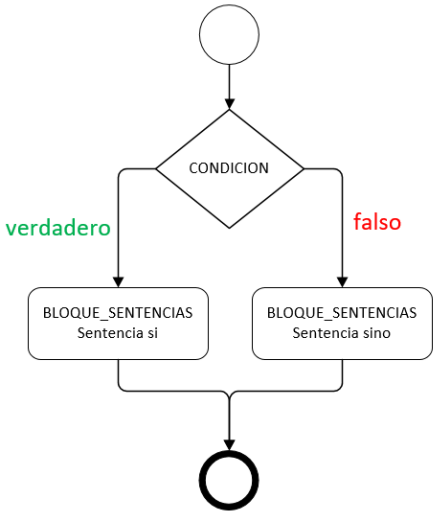


*Ilustración 2: diagrama de flujo para la sentencia de selección **si***

Selecciona

Esta sentencia de selección será la bifurcación múltiple, es decir, proporcionará control sobre múltiples alternativas basadas en el valor de una expresión de control.

La sentencia **selecciona** compara con el valor seguido de cada **caso**, y si coincide, se ejecuta el bloque de sentencias asociadas a dicho **caso**, hasta encontrar una instrucción **detener**.

SINTAXIS

```
selecciona(EXPRESION) {
caso EXPRESION: BLOQUE_SENTENCIAS [detener;]
(caso EXPRESION: BLOQUE_SENTENCIAS [detener;])*
[defecto: BLOQUE_SENTENCIAS [detener;]]
}
```

EJEMPLO

```
selecciona (y /50) {
     caso 1: {
     // sentencias que se ejecutarán si y/50 = 1.
     } caso 2: {
     // sentencias que se ejecutarán si y/50 = 2.
     } default: {
     // sentencias que se ejecutarán si y/50 no coincide ningún caso.
     }
}
```

NOTAS

- Si i es el caso actual y n el número de casos, si la expresión del caso i llegara a coincidir con la expresión de control (para $i < n$), se ejecutarán todos los bloques de sentencias entre los casos $i\ y\ n$ mientras no se encuentre la instrucción **detener**.
- Tanto la expresión de control como las expresiones asociadas a cada uno de los casos deberá ser de tipo primitivo, es decir, `carácter, entero, decimal` o `booleano`.
- Si ninguno de los casos coincide con la expresión de control se ejecutará el bloque de sentencias asociadas a **defecto.**

4.3.26 Sentencias cíclicas o bucles

Mientras

La sentencia cíclica mientras se utilizará para crear repeticiones de sentencias en el flujo del programa.

El bloque de sentencias se ejecutará mientras la condición sea verdadera $(0\ a\ n\ veces)$, de lo contrario el programa continuará con su flujo de ejecución normal.

SINTAXIS

```
mientras(CONDICION) BLOQUE_SENTENCIAS
```

EJEMPLO

```
entero cuenta = 0;
mientras (cuenta <= 10) {
      cuenta++;
}
```

Hacer-Mientras

La sentencia cíclica hacer-mientras se utilizará para crear repeticiones de sentencias en el flujo del programa.

El bloque de sentencias se ejecutará una vez y se seguirá ejecutando mientras la condición sea verdadera $(1\ a\ n\ veces)$, de lo contrario el programa continuará con su flujo de ejecución normal.

SINTAXIS

```
hacer BLOQUE_SENTENCIAS mientras(CONDICION);
```

EJEMPLO

```
entero cuenta = 0;
hacer {
      cuenta++;
} mientras (cuenta <= 10);
```

Para

La sentencia cíclica para permitirá inicializar o establecer una variable como variable de control, el ciclo tendrá una condición que se verificará en cada iteración, luego se deberá definir una operación que actualice la variable de control cada vez que se ejecuta un ciclo para luego verificar si la condición se cumple.

SINTAXIS

```
para(DECLARACION | ASIGNACION; CONDICION ; ACTUALIZACION)
BLOQUE_SENTENCIAS
```

EJEMPLO

```
para (entero cuenta = 0; cuenta <= 10; cuenta++) {
      a[cuenta] = cuenta;
}
```

Inicializacion: Declaración o asignación.

Condicion: Expresión booleana que se evaluara cada iteración para decidir si se ejecuta el bloque de sentencias o no.

Actualizacion: Sentencia que actualiza la variable de control ya sea con una asignación o una expresión de incremento o decremento

4.3.27 Estructuras de Datos

El lenguaje Coline, tendrá estructuras de datos predefinidas, las cuales ayudarán a un mejor el manejo de información de forma dinámica, de esta forma el lenguaje no se limitará únicamente a la utilización de vectores.

4.3.27.1 Listas

Las listas de datos son estructuras que podrán contener diferentes tipos de datos, estas crecerán o disminuirán su tamaño según sea necesario. Las listas utilizarán la palabra reservada lista y una vez sean instanciadas estas se encontrarán vacías. Al momento de inicializar una lista se deberá de indicar el tipo de dato o clase, para la lista, por lo que esta solo podrá contener elementos de este tipo.

Sintaxis

```
Lista identificador = nuevo Lista(Tipo_de_dato);
```

Ejemplo

```
Lista miLista = nuevo Lista(Entero);
```

Insertar

Este método permitirá insertar un elemento en la lista de datos, al momento de insertar un elemento este será insertado al final de la lista. Los elementos insertados deberán corresponder al tipo de elemento predefinido al momento que se instancio la lista.

Sintaxis

```
Identificador.insertar(Elemento);
```

Ejemplo

```
Lista miLista = nuevo Lista();

miLista.insertar(10); //Lista = 10->
miLista.insertar(50); //Lista = 10->50->
miLista.insertar(100); //Lista = 10->50->100->
```

Acceso a elementos

El acceso a elementos de la lista se podrá realizar a través del nombre de la lista y la utilización de un índice numérico. El índice deberá ir dentro de llaves seguido del nombre de la lista. Si el índice esta fuera de la cantidad de elementos de la lista devolverá un resultado nulo. Los índices iniciarán en cero.

Sintaxis

```
Identificador.obtener(índice);
```

Ejemplo

```
Lista miLista = nuevo Lista();
Entero miNumero;

miLista.insertar(10); //Lista = 10->
miLista.insertar(50); //Lista = 10->50->
miLista.insertar(100); //Lista = 10->50->100->
```

```
miNumero = miLista.obtener(2); //miNumero=100
```

Buscar

Esta sentencia buscará por elemento en la lista devolviendo el índice del mismo dentro de la lista. Si el elemento no se encontrase dentro de la misma devolverá un valor nulo.

Sintaxis

```
Identificador.buscar(expresion);
```

Ejemplo

```
Lista miLista = nuevo Lista();
Entero miIndice;

miLista.insertar(10); //Lista = 10->
miLista.insertar(50); //Lista = 10->50->
miLista.insertar(100); //Lista = 10->50->100->
Entero buscado = 50;
miIndice = miLista.buscar(buscado); //miIndice=1
```

4.3.27.2 Pila

La pila es una estructura de datos que agrega los elementos que al inicio de la estructura. En Coline se utilizará esta estructura la cual únicamente permitirá insertar y sacar elementos al tope de la misma. Cuando sea realizada una instancia de esta estructura de datos, se deberá de indicar el tipo de dato o clase que se ingresara en la misma.

Sintaxis

```
Pila identificador = nuevo Pila(Tipo_de_dato);
```

Ejemplo

```
Pila miPila = nuevo Pila(Entero);
```

Apilar

Este método será utilizado para insertar los elementos a la pila, cada elemento nuevo será ingresado al tope de la pila. El método necesitará la palabra reservada **Apilar** seguida de paréntesis y el elemento a ser apilado.

Sintaxis

```
Identificador.Apilar();
```

Ejemplo

```
Pila miPila = nuevo Pila(Entero);

miPila.Apilar (10); //Pila = 10->
miPila.Apilar (50); // Pila = 50->10->
miPila.Apilar(100); // Pila = 100->50->10->
```

Desapilar

Este método será utilizado para sacar los elementos a la pila, cada elemento desapilado será el que se encuentre al tope de la pila. El método solo necesitará la palabra reservada **Desapilar** seguido de paréntesis.

Sintaxis

```
Identificador.Desapilar();
```

Ejemplo

```
Pila miPila = nuevo Pila(Entero);

miPila.Apilar (10); //Pila = 10->
miPila.Apilar (50); // Pila = 50->10->
miPila.Apilar(100); // Pila = 100->50->10->

miPila.Desapilar(); // Pila = 50->10->
```

4.3.27.3 Cola

La cola es una estructura de datos que agrega los elementos al final de la estructura. En Coline se utilizará esta estructura la cual únicamente permitirá insertar datos al final de la estructura, y sacar elementos al inicio de la misma. Cuando sea realizada una instancia de esta estructura de datos, se deberá de indicar el tipo de dato o clase que se ingresara en la misma.

Sintaxis

```
Cola identificador = nuevo Cola(Tipo_de_dato);
```

Ejemplo

```
Cola colaNueva = nuevo Cola(Entero);
```

Encolar

Este método será utilizado para insertar los elementos a la cola, cada elemento nuevo será ingresado al final de la cola. El método necesitará la palabra reservada **Encolar**, seguido de paréntesis y el elemento a insertado al final.

Sintaxis

```
Identificador.Encolar();
```

Ejemplo

```
Cola colaNueva = nuevo Cola(Entero);

colaNueva.Encolar (10); //Cola = 10->
colaNueva.Encolar(50); // Cola = 10->50->
colaNueva.Encolar(100); // Cola = 10->50->100->
```

Desencolar

Este método será utilizado para sacar los elementos de la cola, cada elemento desencolado será el que se encuentre al inicio de la cola. El método solo necesitará la palabra reservada **Desencolar** seguido de paréntesis.

Sintaxis

```
Identificador.Desencolar();
```

Ejemplo

```
Cola colaNueva = nuevo Cola();

colaNueva.Encolar (10); //Cola = 10->
colaNueva.Encolar(50); // Cola = 10->50->
colaNueva.Encolar(100); // Cola = 10->50->100->

colaNueva. Desencolar (); // Cola = 10->50->100->
```

```
Entero miIndice;
```

4.3.27.4 Mostrar EDD

Esta sentencia buscará por elemento en la lista devolviendo el índice del mismo dentro de la lista. Si el elemento no se encontrase dentro de la misma devolverá un valor nulo.

Sintaxis

```
MostrarEDD(id_estructura);
```

Ejemplo

```
Lista miLista = nuevo Lista();
miLista.insertar(10); //Lista = 10->
miLista.insertar(50); //Lista = 10->50->
miLista.insertar(100); //Lista = 10->50->100->
MostrarEDD(miLista);
// Salida en consola: inicio=>> 10->50->100 <<=fin

Pila miPila = nuevo Pila(Entero);
miPila.Apilar (10); //Pila = 10->
miPila.Apilar (50); // Pila = 50->10->
miPila.Apilar(100); // Pila = 100->50->10->
miPila.Desapilar(); // Pila = 50->10->
MostrarEDD(miPila);
// Salida en consola: tope=>> 50->10 <<=fondo

Cola colaNueva = nuevo Cola();
colaNueva.Encolar (10); //Cola = 10->
colaNueva.Encolar(50); // Cola = 10->50->
colaNueva.Encolar(100); // Cola = 10->50->100->
colaNueva. Desencolar (); // Cola = 10->50->100->
MostrarEDD(colaNueva);
// Salida en consola: inicio=>> 10->50->100 <<=fin
```

4.3.28 Lectura de datos de parte del usuario

En Coline existirá una función que interrumpirá el flujo normal del programa para pedir un valor al usuario, a través de una ventana emergente (popup) mostrando un mensaje en la ventana, valor ingresado el cual deberá ser almacenado en la variable identificada por el primer parámetro. Si el tipo de dato recibido es diferente del dato esperado deberá de mostrar un mensaje de error y volverá a solicitarlo.

Sintaxis

```
Leer_Teclado (cadena_mensaje, variable);
```

Ejemplo

```
Entero  Id_1;
Caracter Id_2[20];
Caracter Id_3;
Booleano Id_4;
Decimal Id_5;

Leer_Teclado ( “Ingrese un número”, Id_1);
Leer_Teclado ( “Ingrese una Cadena”, Id_2);
Leer_Teclado ( “Ingrese un carácter”, Id_3);
Leer_Teclado ( “Ingrese un Verdadero o verdadero”, Id_4);
Leer_Teclado ( “Ingrese un número decimal”, Id_5);
```

5 El formato de código intermedio

El formato de código intermedio que se tendrá que generar es el código de tres direcciones, se le llama “de tres direcciones” porque todas sus sentencias deben tener a lo sumo tres posiciones en memoria o valores, es decir que cualquier operación compuesta debe ser “separada” en operaciones que cumplan con dicha premisa, en esta sección se describirán los detalles del código intermedio a generar:

5.1 Comentarios

Para mejorar la legibilidad, y poder realizar comprobaciones sobre el código de mejor manera se define la posibilidad de agregar comentarios al código intermedio, estos comentarios pueden ser de una sola línea, empezarán con **//** y terminarán con un salto de línea; o bien pueden ser comentarios multilínea, que inicien con **/***y terminen con ***/**.

Ejemplos:

```
// Este es un comentario de una línea

/* Este en cambio, es

     un comentario

     múltiples

     líneas

*/
```

5.2 Temporales

Un temporal o variable auxiliar es una variable creada en la generación de código de tres direcciones para almacenar el resultado de una operación. El nombre propio de un temporal está compuesto por una letra t seguida de cualquier número entero. Es necesario tomar en cuenta que el índice que da nombre a los temporales debe aumentar independientemente de los métodos en los que los temporales sean utilizados.

```
     Expresión regular: t[0-9]+
```

Ejemplo:

```
t12
t69
```

5.3 Etiquetas

Una etiqueta es la identificación de un lugar específico dentro del código de tres direcciones, la cual puede trasladar la ejecución del código por medio de algún salto de manera explícita. Una etiqueta está compuesta por la letra L seguida de cualquier número entero. Es necesario tomar en cuenta que el contador debe aumentar independientemente de los métodos..

```
Expresión regular: l[0-9]+
```

Ejemplo:

```
l12
l69
```

5.4 Proposición de asignación

Las proposiciones de asignación serán utilizadas para poder asignar un valor a una variable (temporal o puntero). Las proposiciones de asignación tendrán la siguiente sintaxis:

Asignación con un operador aritmético

```
<id> = <id_o_valor> <operador> <id_o_valor>;
```

Asignación simple

```
<id> = <id_o_valor>;
```

Donde:

- **<id>**es el destino del valor que se encuentra al lado derecho de la asignación
- **<id_o_valor1>**y **<id_o_valor2>**pueden ser temporales, punteros, o valores fijos
- **<operador>**debe ser exclusivamente un operador ARITMÉTICO

Ejemplo:

```
t1 = 1990.0620;
t2 = 10;

t3 = t1 + t2;
t4 = 5 * t4;
```

5.5 Operaciones aritméticas

Las operaciones aritméticas que se realizarán dentro del código de tres direcciones serán las operaciones más básicas, las cuales se describen en la siguiente tabla:

Operación	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/
Módulo	%

5.6 Operaciones relacionales

Las operaciones relacionales que se realizarán dentro del código de tres direcciones serán las operaciones más básicas, las cuales se describen en la siguiente tabla:

Operaciones	Símbolo
Comparación	==
Diferencia	!=
Mayor	>
Mayor o igual	>=
Menor	<
Menor o igual	<=
Verdadero	1==1
Falso	1==0

5.7 Operaciones lógicas

El código intermedio de las operaciones lógicas será escrito en formato de corto circuito. Por lo tanto, NO es posible utilizar los operadores que normalmente se utilizan para dichas operaciones.

5.8 Salto incondicional

Este salto **NO** depende del cumplimiento de una condición para poder realizarse. El salto incondicional sigue la siguiente sintaxis:

```
goto <etiqueta>;
```

Ejemplo:

```
void metodo_1(){
	L1:
	t1 = 2;
	t2 = 5.89;

	goto L2; // Este es un salto incondicional t3 = S + 0;
	t4 = 10;
	t5 = 3.14 * t4;
	L2:

	//Instrucciones para la etiqueta L2

}
```

5.9 Salto condicional

La ejecución de este salto, depende del cumplimiento de la condición señalada por el `if` , si dicha condición se cumple, el salto SI debe realizarse, de lo contrario, la ejecución continuará en la siguiente línea de código. El salto condicional sigue la siguiente sintaxis.

Ejemplo:

```
if (<id_o_valor1> <op_rel> <id_o_valor2>) goto <etiqueta>;
```

Donde:

- **<op_rel>**representa cualquiera de los operadores relacionales
- **<id_o_valor1>**y **<id_o_valor2>**pueden ser temporales, punteros, o valores fijos
- **<etiqueta>** representa el lugar dentro del código al cual se trasladará la ejecución en caso de que sí se cumpla la condición

5.10 Declaración de métodos

Dentro del código de tres direcciones un método será declarado por medio de la siguiente sintaxis, cabe resaltar que TODOS los métodos son de tipo **void** y se permiten llamadas recursivas.

```
void identificador(){
    //Instrucciones en código de 3 direcciones
}
```

Ejemplo:

```
void metodo_1(){
    //Instrucciones en código de 3 direcciones

}
```

5.11 Llamadas a métodos

Mientras que para las llamadas a métodos se utilizará la siguiente sintaxis, en este nivel, los métodos no admiten parámetros, ya que los mismos se trasladan por medio del Stack.

```
<id_metodo>();
```

Ejemplo:

```
void metodo_1(){
    // Esta es una llamada al metodo_2 dentro de
    metodo_1 metodo_2();
}

void metodo_2(){
    // Instrucciones en código de 3 direcciones

}
```

5.12 Funciones nativas del código intermedio

Como en todo lenguaje de programación, para soportar las funciones primitivas del lenguaje de alto nivel es necesario definir un core de funciones nativas.

5.12.1 Core

Coline incorpora un repertorio de funciones nativas de código de tres direcciones para apoyar al lenguaje de alto nivel, el desarrollo del cuerpo de las instrucciones de este core queda a discreción del estudiante, pero deben tener los siguientes nombres:

Función nativa	Parámetros en la pila
$$_getBool()	[0 | retorno (0 = F, 1 = V) [1 | referencia a la cadena a convertir]
$$_getNum()	[0 | retorno, valor numérico] [1 | referencia a la base] [2| referencia a la cadena a convertir] [3| valor numérico por defecto]
$$_outStr()	[0| retorno, no se usa] [1| referencia a la cadena a mostrar]
$$_outNum()	[0 | retorno, no se usa] [1 | valor numérico] [2 | valor booleano (0 = F, 1 = V)]
$$_inStr()	[0 | retorno, no se usa] [1 | referencia donde guardar el valor] [2 | referencia a la cadena a mostrar]
$$_inNum()	[0 | retorno, valor numérico] [1 | referencia a la cadena a mostrar] [2 | valor numérico por defecto]
$$_show()	[0 | retorno, no se usa] [1 | referencia a la cadena a mostrar]
$$_getRandom	[0 | retorno, valor numérico]
$$_getArrLength	[0 | retorno, valor numérico] [1| referencia al arreglo] [2| valor numérico, dimensión]
$$_getStrLength	[0| retorno, valor numérico] [1| referencia de la cadena a analizar]

Nota: * Como no se pueden redefinir los arreglos es posible que las llamadas a esta función sean reemplazadas en tiempo de compilación por el valor consultado directamente desde la tabla de símbolos, asumiendo que el segundo parámetro únicamente puede ser un valor numérico (entero) primitivo, es decir, no expresiones o variables.

Para llamar a cualquiera de estas funciones es necesario cargar previamente los parámetros en la pila para seguidamente proceder a realizar la llamada a la función y luego sustraer el valor retornado según sea el caso.

5.12.2 Printf

El código de 3 direcciones contará con una función para poder imprimir salidas en consola, para ello se utilizará la palabra clave “printf” con la sintaxis que se muestra a continuación:

```
printf(<param> , <id_o_valor>);
```

Donde:

- **<param>**indicará el formato en que se mostrará el valor
- **<id_o_valor>**representa el valor que se mostrará

Parámetro	Formato según el valor de param
%c	Indica que el valor a imprimir será un carácter.
%d	Indica que el valor a imprimir será un entero.
%f	Indica que el valor a imprimir será un número con punto flotante.

5.12.3 Exit

Este método es la forma en que se traducen las excepciones al código intermedio. Las llamadas a este método tienen un código que indica cuál es el error que se ha suscitado, al finalizar la ejecución de todo el programa debe llamarse a este método con un 0 como valor de su parámetro, esto indicará que la ejecución ha terminado correctamente.

6 Entorno de ejecución

Como ya se expuso en otras secciones de este documento, en el código intermedio **NO** existen cadenas, operaciones complejas, llamadas a métodos con parámetros y muchas otras cosas que sí existen en los lenguajes de alto nivel. Esto debido a que el código intermedio busca ser una representación próxima al código máquina, por lo que todas las sentencias de las que se compone se deben basar en las estructuras que componen el entorno de ejecución. Típicamente se puede decir que los lenguajes de programación cuentan con dos estructuras para realizar la ejecución de sus programas en bajo nivel, la pila (Stack) y el montículo (Heap), en la siguiente sección se describen estas estructuras.

6.1 Estructuras del entorno de ejecución

Las estructuras del entorno de ejecución no son más que arreglos de bytes que emplean ingeniosos mecanismos para emular las sentencias de alto nivel. En este proyecto las estructuras de control serán representadas por arreglos de números con punto flotante, esto con el objetivo de que se pueda tener un acceso más rápido a los datos sin necesidad de leer por grupos de bytes y convertir esos bytes al dato correspondiente, como normalmente se hace en otros lenguajes, por ejemplo en Java, un int ocupa 4 bytes, un boolean ocupa 1 solo byte, un double ocupa 8 bytes, un char 2 bytes, etc.

6.1.1 El Stack y su puntero

El stack es la estructura que se utiliza en código intermedio para controlar las variables locales, y la comunicación entre métodos (paso de parámetros y obtención de retornos en llamadas a métodos). Se compone de ámbitos, que son secciones de memoria reservados exclusivamente para cierto grupo de sentencias.

Cada llamada a método o función que se realiza en el código de alto nivel cuenta con un espacio propio en memoria para comunicarse con otros métodos y administrar sus variables locales. Esta estructura recibe su nombre debido al comportamiento que presenta para la administración de los ámbitos, puesto que utiliza la política LIFO(Last In First Out) para apilar los ámbitos uno sobre otro y conforme la ejecución de un método termina se va liberando el espacio reservado para éste, eliminando el ámbito que se encuentra en la cima de la pila. Más información de LIFO en https://es.wikipedia.org/wiki/Last_in,_first_out.

Esto se logra modificando el puntero del Stack, que en el proyecto se identifica con la letra P, para ir moviendo el puntero de un ámbito a otro, cuidando de no corromper ámbitos ajenos al que se está ejecutando. A continuación, se ilustra su funcionamiento con un ejemplo.

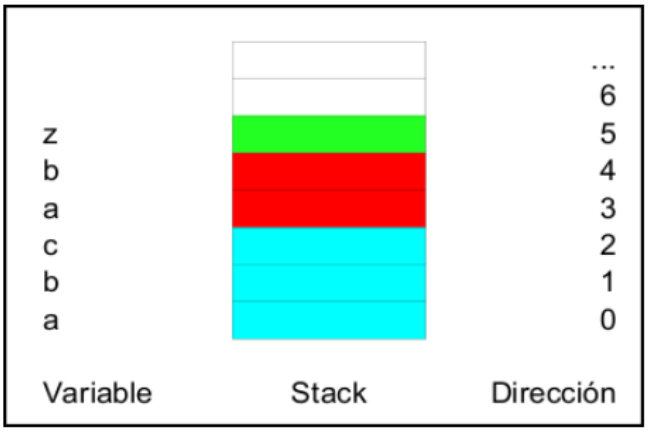


Imagen 21: Stack

Cabe resaltar que por su misma naturaleza (LIFO) el Stack se reutiliza por lo que una buena práctica es limpiar el área que ya no se utilizará, colocando valores nulos en todo el espacio que ha dejado libre la finalización de un método, para ello se cuenta con la instrucción nativa.

$$_SGB(num inicio, num longitud)

Se debe llamar al terminar una llamada a un método y se le deben enviar como parámetros el valor de inicio de un ámbito y el tamaño del mismo para realizar la limpieza del espacio ocupado por el método. Esta instrucción no será programada en el core de funciones nativas, más sí tendrá que ser ejecutada por el intérprete de código de tres direcciones. La colocación de esta instrucción debe estar en la propuesta del código intermedio que realice el estudiante para las llamadas a métodos, como sugerencia se podría tener la siguiente plantilla para las llamadas a métodos o funciones.

```
tx = P + K;          % Simulación de cambio de ámbito, K = tamaño

                     %      Carga      de
 . . .               parámetros          (si aplica)

 . . .               % Sigue la carga de parámetros

                     %  Termina  la  carga
 . . .               de                  parámetros

P =  P + K;          % Cambio real del ámbito
```

```
metodo_funcion();  % Llamada al método
                   %
. . .              Obtención  del retorno (si aplica)
P = P - K;         % Regreso del ámbito
$$_SGC(tx, K);     % Llamada a SGC para limpiar el ámbito terminado
```

6.1.2 El Heap y su puntero

El Heap (o en español, montículo) es la estructura de control del entorno de ejecución encargada de guardar las estructuras o referencias a variables globales. Esta es una estructura que también puede almacenar basura digital (datos inalcanzables) pero la implementación de un garbage collector para este tipo de estructuras requiere un esfuerzo mucho mayor que no realizaremos en este proyecto.

El puntero H, a diferencia de P (que aumenta y disminuye según lo dicta el código intermedio), solo aumenta y aumenta, su función es brindar siempre la próxima posición libre de memoria. A continuación se muestra un ejemplo de cómo se vería el Heap después de almacenar dos estructuras (element), una de tipo **`Arbol`** y otra de tipo **`Nodo`** , además de poder ver que la próxima posición que nos brindaría el Heap para almacenar las estructuras sería la 12, el valor de su puntero H.

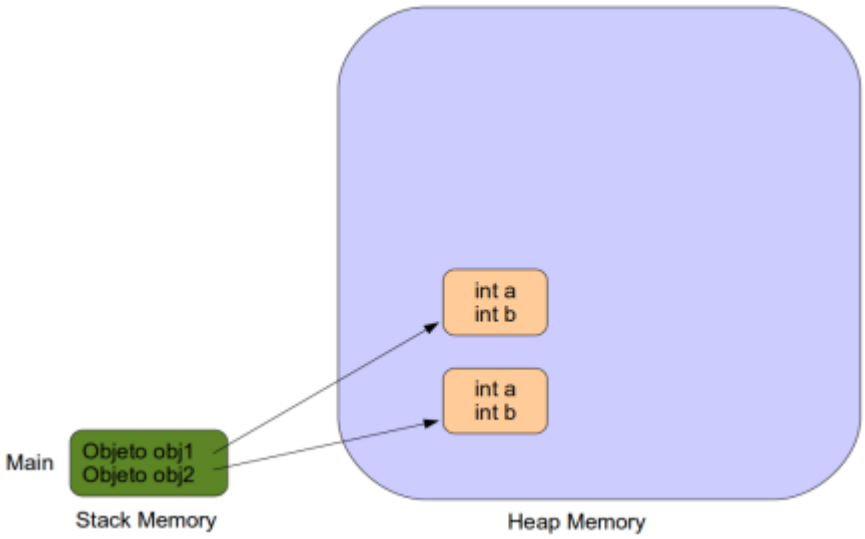


Imagen 22: Heap

Llegados a este punto, puede que exista el cuestionamiento de ¿Dónde están almacenadas las cadenas de texto? para eso en este proyecto se implementará una estructura adicional en el entorno de ejecución, una estructura llamada String Pool, cuyo único objetivo es el de almacenar cadenas de texto, esta estructura almacenará exclusivamente los códigos ascii de cada uno de los caracteres que componen una cadena.

En la siguiente subsección se describe el uso y otros aspectos de esta estructura.

6.1.3 El String Pool y su puntero

Esta estructura está dedicada a almacenar únicamente cadenas de caracteres (texto) cada cadena almacenada debe terminar con el caracter de final de cadena '\0', al que le corresponde el código ascii 0, así pues se tendrá un forma fácil de observar las cadenas almacenadas desde la vista de debug.

El String Pool será accesible por medio del identificador Pool y su puntero estará identificado por la letra S y éste, al igual que H, solo podrá ir en aumento, otorgando siempre la próxima posición disponible en memoria. Para almacenar la cadena "USAC" se debería generar algo como lo siguiente:

```
% Almacenando USAC
tx = H;                 % Copia de H
H = H + 1;              % Aumento de H
Heap[tx] = S;           % Almacenando referencia inicial a String Pool
Pool[S] = 85;           % Copia 'U', ascii 85
S = S + 1;              % Aumenta S
Pool[S] = 83;           % Copia 'S', ascii 83
S = S + 1;              % Aumenta S
Pool[S] = 65;           % Copia 'A', ascii 65
S = S + 1;              % Aumenta S
Pool[S] = 67;           % Copia 'C', ascii 67
S = S + 1;              % Aumenta S
Pool[S] = 0;            % Copia del caracter de fin '\0', ascii 0
S = S + 1;              % Aumenta S
```

En el manejo de las cadenas siempre es importante realizar una **“doble referencia”**, esto quiere decir que cualquier cadena siempre tendrá primero una posición en el Heap, en la que pueden almacenarse dos posibles cosas, se almacenará la posición de inicio de la cadena real (el primer caracter) dentro del String Pool o se almacenará NULL, con lo primero se garantiza de que toda cadena tenga una referencia válida, y con lo segundo se garantiza que dicha referencia pueda apuntar a NULL que sería el caso en el que no se ha inicializado la cadena o bien se hizo una asignación de **`cadena=Nada`**.

String Pool S=60

POSICIÓN	VALOR
0	Esta cadena ocupa 29 espacios
30	Cadena de prueba
46	Compiladores 2
60	

6.2 Acceso a estructuras del entorno de ejecución

Para asignar un valor a una estructura del sistema es necesario colocar el identificador del arreglo (Stack, Heap o Pool), la posición donde se desea colocar el valor seguido del valor a asignar con la siguiente sintaxis:

```
Stack[id_o_valor] = id_o_valor;
// o bien
Heap[id_o_valor] = id_o_valor;
// o bien
Pool[id_o_valor] = id_o_valor;
```

Para la obtención de un valor de cualquiera de las estructuras (Stack, Heap o Pool) se realizará con la siguiente sintaxis:

```
id = Stack[id_o_valor];
// o bien
id = Heap[id_o_valor];
// o bien
id = Pool[id_o_valor];
```

7 Optimización de código intermedio

La optimización de código consiste en mejorar el rendimiento y la eficiencia del programa compilado. La optimización de código debe realizarse mediante la unión de la optimización por bloques y la optimización por mirilla.

Primero se deberá aplicar las reglas de optimización de flujo de control sobre todo el código de entrada, luego se realizará la unión de las optimizaciones (bloques y mirilla) aplicando el concepto de construcción de bloques, luego la optimización se hará utilizando unas de las reglas de manera local de la optimización por bloques y para cada bloque se aplicarán las reglas de la optimización por mirilla

Para la construcción de los bloques, se debe considerar los líderes de los bloques que no son más que la instrucción con la que se comienza dentro del bloque, las reglas para buscar líderes son:

1. La primera instrucción de tres direcciones en el código intermedio
2. Cualquier instrucción que sea el destino de un salto condicional o incondicional
3. Cualquier instrucción que siga justo después de un salto condicional o incondicional

Con las reglas establecidas se puede establecer que para cada instrucción líder, su bloque básico consiste en sí misma y en todas las instrucciones hasta, pero sin incluir a la siguiente instrucción líder o el final del programa intermedio (Ver sección 8.4 del libro de clase).

Como principio de la optimización por bloques se deberán establecer relaciones entre los bloques, las cuales consisten en apuntadores de un bloque a otro y que existen para ver el flujo que contiene la ejecución del código intermedio.

Después de tener establecido los bloques, se debe aplicar las transformaciones de **manera local** sobre los bloques y obtener como salida el código optimizado. Las transformaciones a considerar son 3, dentro de las cuales se definirán 16 reglas para optimizar el código.

- Eliminación de subexpresiones comunes y propagación de copias
- Simplificación algebraica
- Reducción por fuerza

7.1 Eliminación de sub expresiones comunes

Regla No. 1

Para eliminar una subexpresión, esta debió ser calculada previamente y los valores que integran la subexpresión no deben cambiar desde el momento que es calculada hasta que se utilice. Ejemplo:

Código generado	Código optimizado
a = b + c b = a - d c = b + c d = a - d	a = b + c b = a - d c = b + c d = b

7.2 Propagación de copias

Regla No. 2

La regla para propagación de copias permitirá eliminar código inactivo o redundante:

Código generado	Código optimizado
t1 = i + 4 i = i + 4 t2 = j - 1 j = j -1	t1 = i + 4 i = t1 t2 = j - 1 j = t2

7.3 Simplificación algebraica

Un optimizador puede utilizar identidades algebraicas para eliminar las instrucciones ineficientes. Para esta transformación definimos las siguientes reglas de optimización:

Regla No. 3

Se eliminará la instrucción que cumpla con el siguiente formato:

```
x = x + 0; % Donde x es cualquier identificador % ó bien
x = 0 + x; % Donde x es cualquier identificador
```

Dado que no cumple ninguna funcionalidad dentro del código.

Regla No. 4

Se eliminará la instrucción que cumpla con el siguiente formato:

```
x = x - 0; % Donde x es cualquier identificador
```

Dado que no cumple ninguna funcionalidad dentro del código.

Regla No. 5

Se eliminará la instrucción que cumpla con el siguiente formato:

```
x = x * 1; % Donde x es cualquier
identificador % ó bien
x = 1 * x; % Donde x es cualquier identificador
```

Dado que no cumple ninguna funcionalidad dentro del código.

Regla No. 6

Se eliminará la instrucción que cumpla con el siguiente formato:

```
x = x / 1; % Donde x es cualquier identificador
```

Dado que no cumple ninguna funcionalidad dentro del código.

Puede que existan instrucciones como las anteriores pero aplicadas con diferentes variables en el destino y en la expresión del lado derecho de la asignación, en este caso la operación se descarta y se convierte en una asignación. Para estos casos se definen las siguientes reglas de optimización:

Regla No. 7

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o
identificador x = y + 0;
% ó bien
x = 0 + y;

% Se sustituirá por:
x = y;
```

Aplicando la propiedad del elemento neutro de la suma.

Regla No. 8

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o
identificador x = y - 0;

% Se sustituirá por:

 x = y;
```

Aplicando la propiedad del elemento neutro de la resta.

Regla No. 9

Si existe una instrucción que cumpla con la forma siguiente:

```
Donde y es cualquier valor o identificador x = y * 1;

     % ó bien
     x = 1 * y;

     % Se sustituirá
     por: x = y;
```

Aplicando la propiedad del elemento neutro de la multiplicación.

Regla No. 10

Si existe una instrucción que cumpla con la forma siguiente:

```
     % Donde y es cualquier valor o
     identificador x = y * 0;

     % ó bien
     x = 0 * y;

     % Se sustituirá
     por: x = y;
```

Aplicando la propiedad del producto cero.

Regla No. 11

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o
identificador x = y / 1;
% Se sustituirá por:
x = y;
```

Aplicando la propiedad del elemento neutro de la división.

Regla No. 12

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o
identificador x = 0 / y;
% Se sustituirá por:
x = 0;
```

Aplicando la propiedad del cociente cero.

Regla No. 13

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o
identificador x = y ^ 0;
% Se sustituirá por:


x = 1;
```

Aplicando la propiedad del exponente 0.

Regla No. 14

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o
identificador x = y ^ 1;

% Se sustituirá por:

x = y;
```

Aplicando la propiedad del exponente 0.

7.4 Reducción por fuerza

De manera similar puede aplicarse una transformación de reducción por fuerza para sustituir operaciones costosas por expresiones equivalentes relativamente más económicas. Para esta clasificación, se definen las siguientes reglas de optimización:

Regla No. 15

Si existe una instrucción de la forma:

```
% Donde y es cualquier valor o
identificador x = y ^ 2;

% Se sustituirá por:

x = y * y;
```

Regla No. 16

Si existe una instrucción de la forma:

```
% Donde y es cualquier valor o
identificador x = y * 2;

% Se sustituirá por:

x = y + y;
```

7.5 Optimizaciones de flujo de control

Como se mencionó, éstas deberán aplicarse antes de la construcción de bloques y la aplicación de las otras reglas. Ésta optimización surge con la existencia de saltos innecesarios que pueden eliminarse, por lo que se definen las siguientes reglas:

Regla No. 17

Si existe un salto incondicional de la forma ***goto Lx*** y existe la etiqueta ***Lx:*** y la primera instrucción después de la etiqueta es otro salto, de la forma ***goto Ly*** podemos modificar el primer salto para que haga referencia a la etiqueta ***Ly:*** y de esta forma ahorrar la ejecución de un salto

Código generado	Código optimizado
goto L1; **<instrucciones>** **L1: goto L2;**	**goto L2;** **<instrucciones>** **L1: goto L2;**

Regla No. 18

Si existe un salto condicional de la forma ***if<cond> goto Lx;*** y la primera instrucción después de la etiqueta es otro salto, de la forma goto ***Ly*** podemos modificar el primer salto para que haga referencia a la etiqueta ***Ly:***y de esta forma ahorrar la ejecución de un salto.

Código generado	Código optimizado
if **t1<t2 goto L1;** **<instrucciones>** **L1:** **goto L2;**	**if** **t1<t2 goto L2;** **<instrucciones>** **L1:** **goto L2;**

8 Reporte de optimización

Después de realizar una optimización, se deberá poder consultar el reporte de optimización con el fin de evaluar las reglas de transformación aplicadas. El reporte deberá estar seccionado según los bloques que lo compongan y deberá mostrar el código original de cada bloque y luego el código optimizado del mismo, también deberá contener el conjunto de reglas aplicadas (si una regla se usa más de una vez, debe mostrarse en el reporte imprimiendola las veces que se aplique). Este reporte deberá visualizarse de la siguiente manera:

Código original	Código optimizado	Reglas aplicadas	No. Bloque

L1:
t1 = t2 + t3;
t2 = t1 - t4;
t3 = t2 + t3;
t4 = t1 - t4;
goto L2;
	L1:
	t1 = t2 + t3;
	t2 = t1 - t4;
	t3 = t2 + 13;
	t4 = t2;
	goto L2;
		Regla No.1	Bloque 1

L4:
t1 = t20;
t6 = t6+0;
t20 = t1;
t8 = t8+0;
	L4:
	t1=t20;
		Regla No. 3
		Regla No. 2
		Regla No. 3
			Bloque 15

9 Entregables y Restricciones

9.1 Entregables

- Código fuente
- Aplicación funcional
- Gramáticas
- Manual técnico
- Manual de usuario

Deberán entregarse todos los archivos necesarios para la ejecución de la aplicación, así como el código fuente, la gramática y la documentación. La calificación del proyecto se realizará con los archivos entregados en la fecha establecida. El usuario es completa y únicamente responsable de verificar el contenido de los entregables. La calificación se realizará sobre archivos ejecutables. Se proporcionarán archivos de entrada al momento de calificación.

9.2 Restricciones

- La aplicación deberá ser desarrollada utilizando el lenguaje C/C++ con el framework Qt.
- Para la generación de analizadores se utilizarán las herramientas Bison y Flex.
- El intérprete de la representación intermedia de cuádruplos deberá de ser desarrollado utilizando la herramienta Bison y Flex.
- Copias de proyectos tendrán de manera automática una nota de 0 puntos y serán reportados a la Escuela de Ingeniería en Ciencias y Sistemas los involucrados.
- Todos los lenguajes implementados no hacen distinción entre mayúsculas y minúsculas, es decir no son case sensitive.

9.3 Requisitos mínimos

Los requerimientos mínimos del proyecto son funcionalidades del sistema que permitirán un ciclo de ejecución básica, para tener derecho a calificación se deben cumplir con lo siguiente:

- Lecciones
- Tipos de lecciones
    - Dummy-Coach
- Pantallas de la Plataforma
    - Pantalla inicio
    - Editor de texto
    - Participar en lección
    - Nueva lección
    - Evaluar tareas
- Ingreso de código de alto nivel libremente (fuera de una lección)
- Debugger incluyendo los reportes de estructuras en tiempo de ejecución y como mínimo la modalidad línea a línea.

- Coline
    - Case sensitive
    - Sobrecarga de métodos
    - Recursividad
    - Tipos de dato
    - Signos de agrupación
    - Comentarios
    - Operaciones (aritméticas, lógicas y relacionales)
    - Declaración y asignación de variables, objetos y arreglos de una dimensión.
    - Nulo
    - Tamaño de un arreglo unidimensional
    - Operaciones con cadenas
    - Casteos
    - Imprimir
    - Clases
    - Procedimientos, funciones y llamadas a los mismos
    - Constructor
    - Sentencias de transferencia
        - Detener
        - Retorno
    - Sentencias de selección
        - Si
        - Selecciona
    - Sentencias cíclicas
    - Lectura de datos de parte del usuario
- Todas las sentencias del formato de código intermedio (tres direcciones)
- Todas las estructuras en tiempo de ejecución.
- Promedio mayor o igual a 61 en evaluaciones presenciales

Fecha de Entrega:

16 de enero 2019 antes de las 23:59 horas

Forma de entrega:

La forma de entrega será virtual, en la tarea de Classroom asignada.

Crédito y agradecimiento a los siguientes tutores académicos que redactaron el enunciado para el segundo semestre de 2017, ya que en base al mismo fue redactado este enunciado:

Juan Ruiz, José Cano, Henry Taracena y Daniel Álvarez