Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de lenguajes y compiladores 2
Primer Semestre 2018
Catedráticos: Ing. Bayron López, Ing. Edgar Sabán
Tutores académicos: Juan Ruiz, José Cano, Henry Taracena y Daniel Álvarez.

GccCompiler

Segundo proyecto de laboratorio

Contenido

Índice de tablas

Índice de imágenes

1 Descripción

Para el segundo proyecto del laboratorio se deberá desarrollar 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 GccCompiler. Esta aplicación deberá ser desarrollarla e implementarla con javascript en el framework Node.Js.

La aplicación de GccCompiler deberá implementar el siguiente lenguaje:

- Compiler: está basado en el lenguaje C++. Este será un tipo de lenguaje multiparadigma, es decir estructurado y además permitirá manejar 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 G-coach que servirá para aprender la sintaxis de Compiler y su respectivo funcionamiento, el segundo será el tipo de lección A-coach que servirá para poder desarrollar algoritmos y solucionar problemas a través del lenguaje Compiler.

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 GccCompiler

GccCompiler, 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 GccCompiler 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.

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

A-coach: será un tipo de lección para aprender y mejorar las habilidades de resolución de problemas por medio del lenguaje Compiler. 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

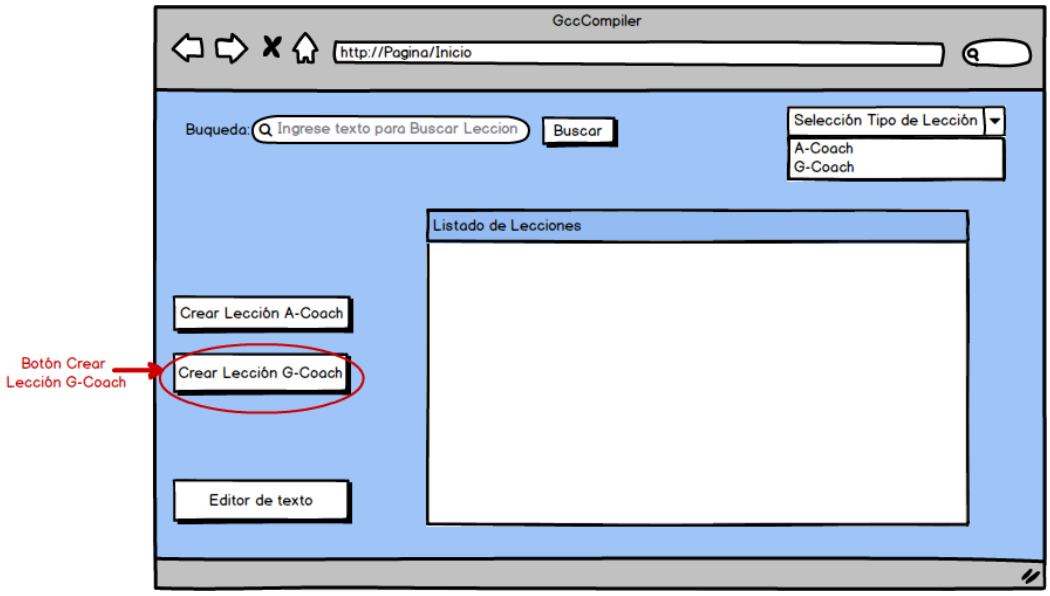
Dentro de la aplicación se podrán realizar las funcionalidades que se describen a continuación.

3.2.1 Crear una nueva lección de tipo G-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 Compiler. Esta lección pertenecerá al tipo G-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 G-coach” como se muestra en la imagen 1.

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



2. Se mostrará la pantalla para el ingreso de una nueva lección de tipo G-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 falsedad de la condición.
   - **Código de ejemplo:**

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

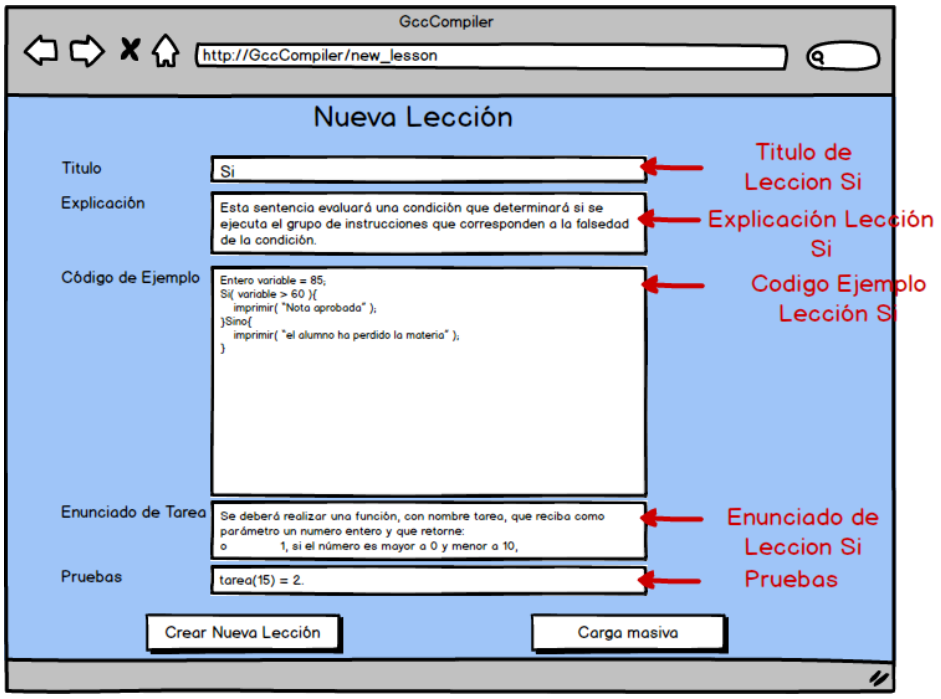
   - **Enunciado de tarea:** se deberá realizar una función, con nombre tarea, que reciba como parámetro un número entero y que retorne:
     - 1, si el número es mayor a 0 y menor a 10,
     - 2, si el número es mayor o igual a 10 y menor a 20.
     - 3, si el número es mayor o igual a 20.

- **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.

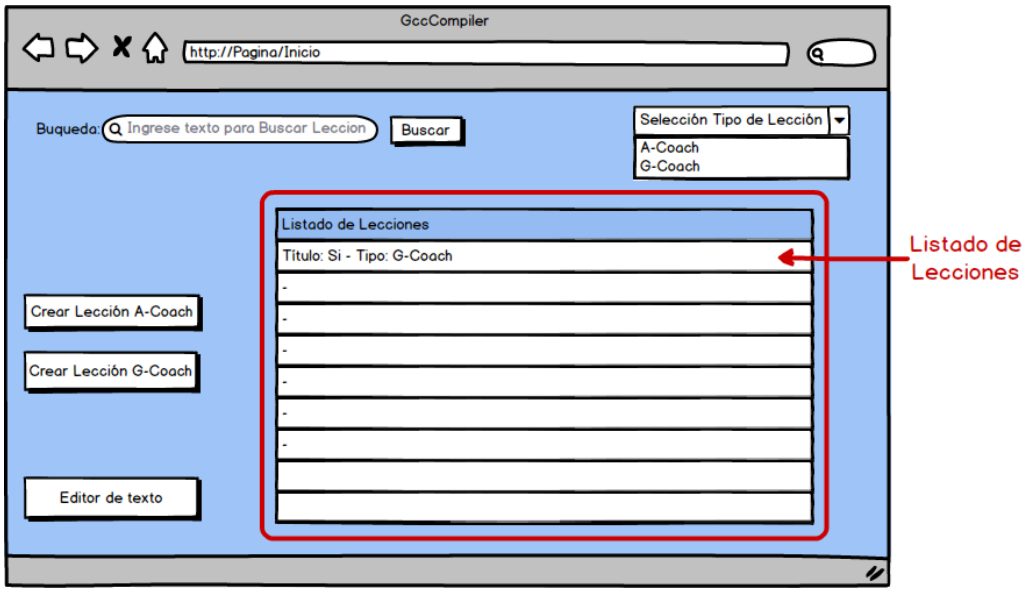
Al llenar los datos de la plataforma se deberá visualizar como se muestra en la imagen 2. Al finalizar de llenar los datos se hará clic en el botón crear nueva lección.

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

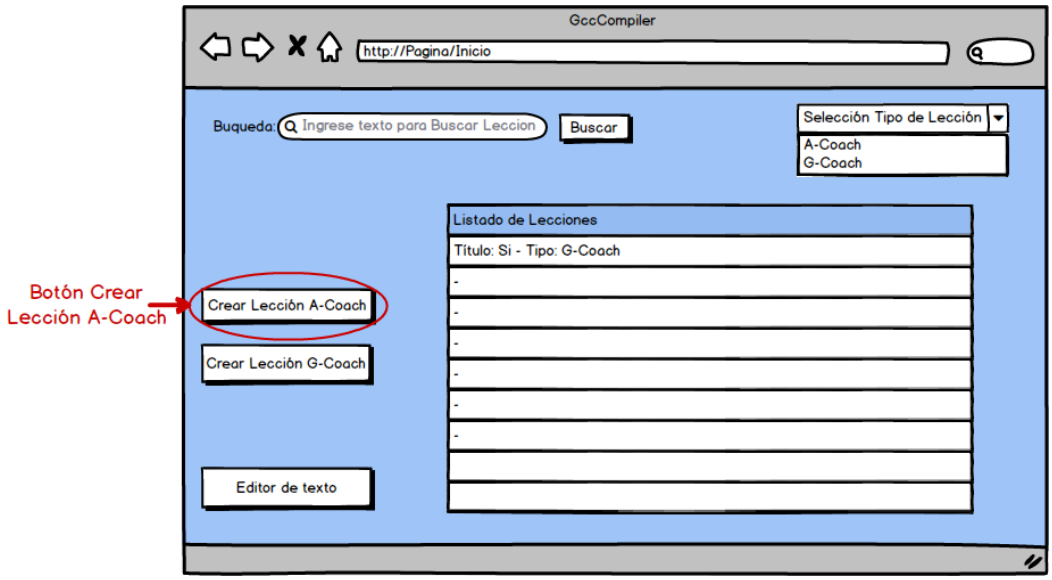


3.2.2 Crear una nueva lección de tipo A-coach

En este caso se desea crear una nueva lección en la aplicación. La lección se enseñará como realizar un ordenamiento en una lista usando el lenguaje Compiler. Esta lección pertenecerá al tipo A-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 A-coach” como se muestra en la imagen 4.

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



4. Se mostrará la pantalla para el ingreso de una nueva lección de tipo A-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:**

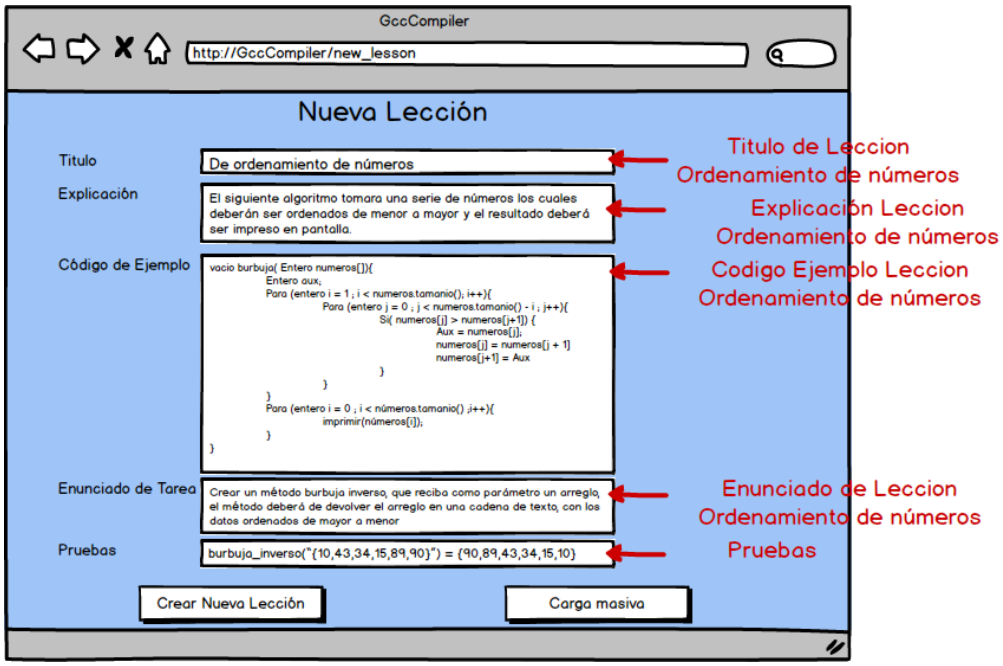
```
vacio burbuja( Entero numeros[]){
        Entero aux;
        Para (entero i = 1 ; i < numeros.tamanio(); i++){
                Para (entero j = 0 ; j < numeros.tamanio() - i ; j++){
                        Si( numeros[j] > numeros[j+1]) {
                                Aux = numeros[j];
      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

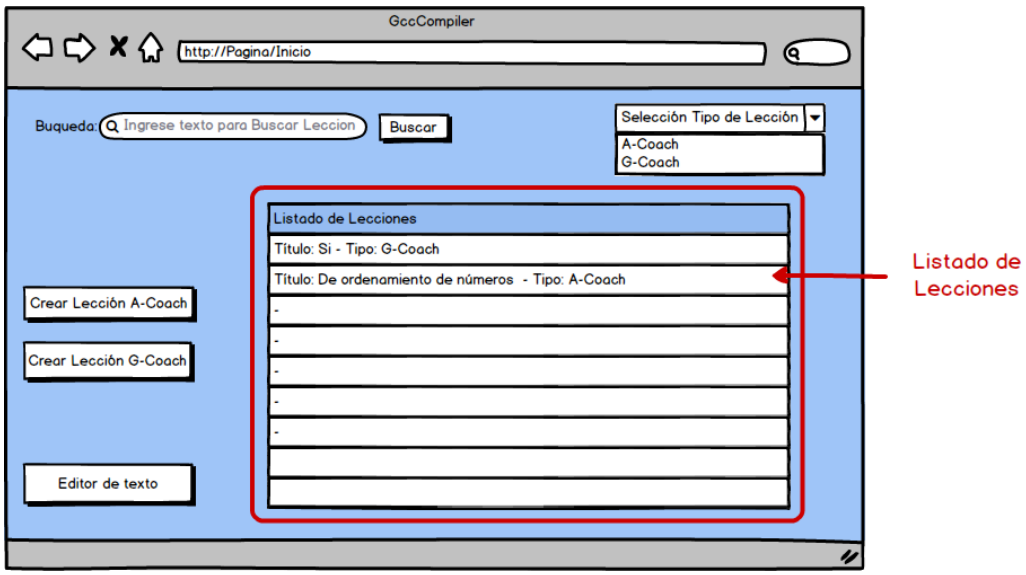
Al llenar los datos de la plataforma se deberá visualizar como se muestra en la imagen 5. Al finalizar de llenar los datos se hará clic en el botón crear nueva lección.

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



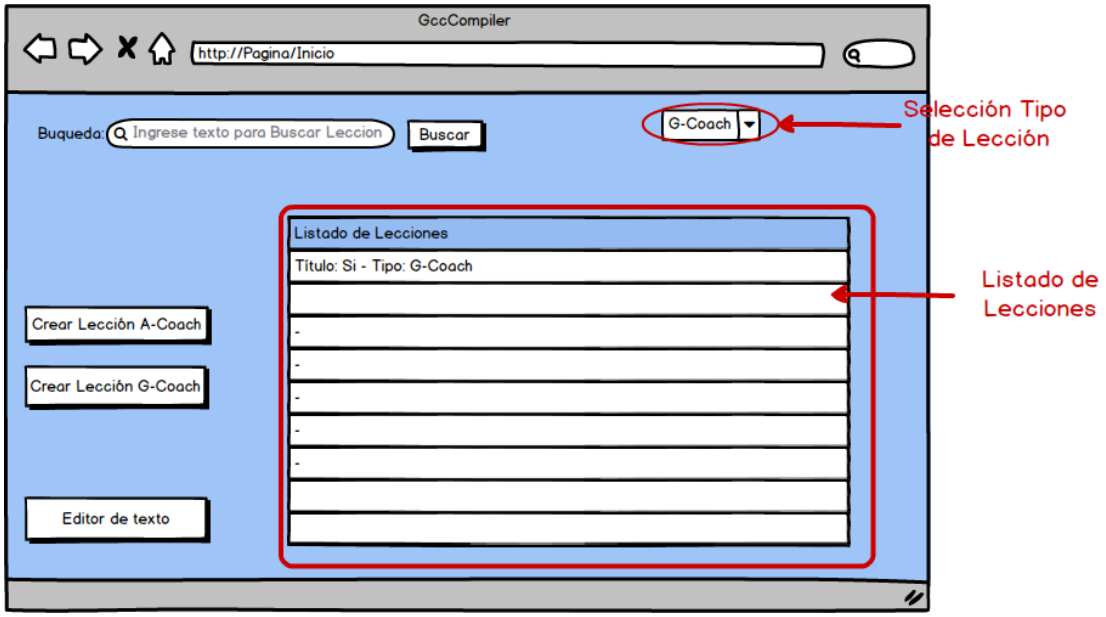
3.2.3 Participar en una lección

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 G-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.

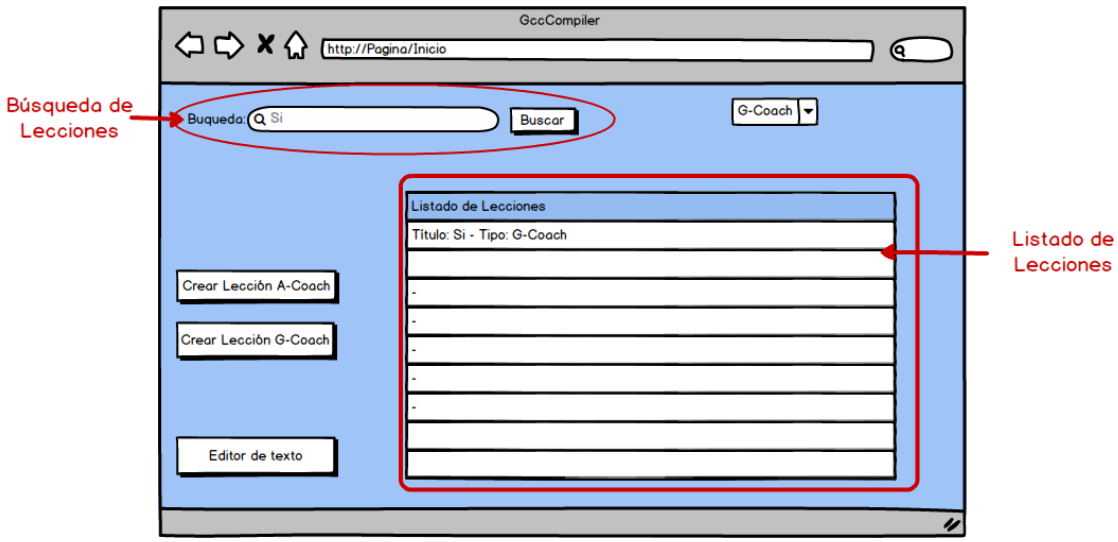
2. Encontrar las lecciones de su interés. Si el usuario desea delimitar el listado de lecciones, el usuario lo podrá hacer por medio de una selección indicando el tipo de lección, que desea ver en el listado tal y como se muestra en la imagen 7.

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



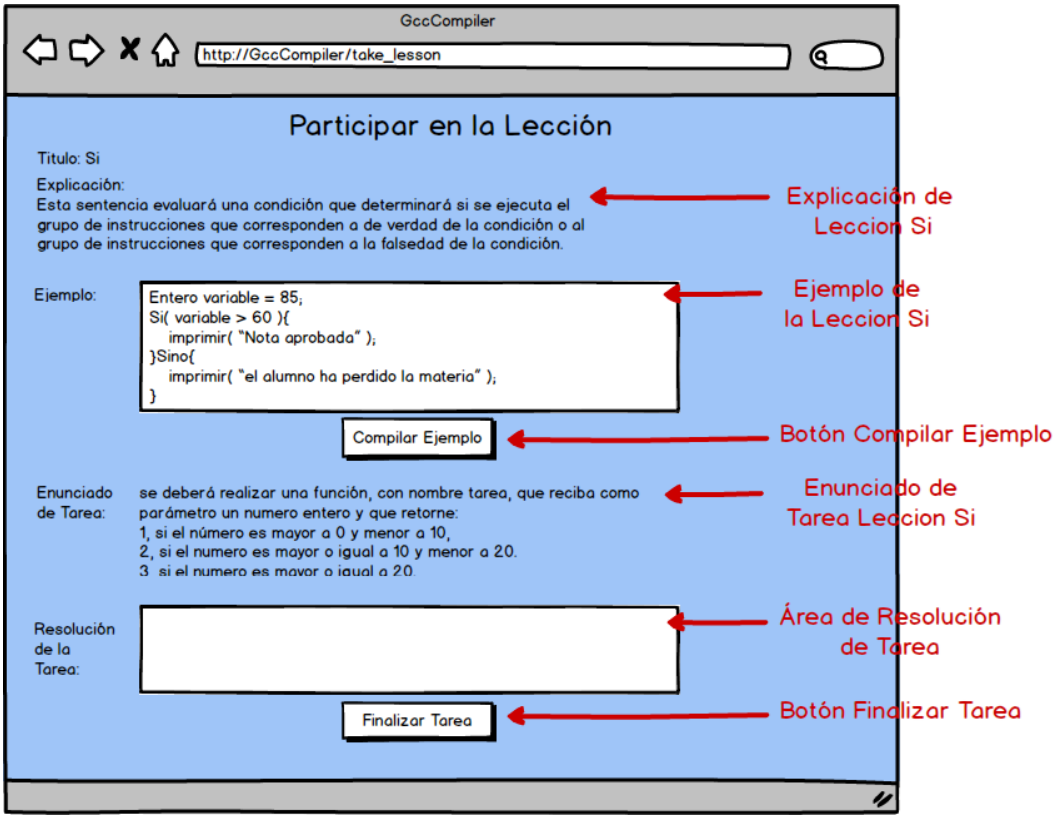
3. Otra opción para delimitar el listado de lecciones es por medio de una búsqueda, que el usuario ingresará el texto a buscar, se mostrarán en el listado de lecciones únicamente los títulos que contengan las palabras ingresadas tal y como se muestra en la imagen 8.

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.

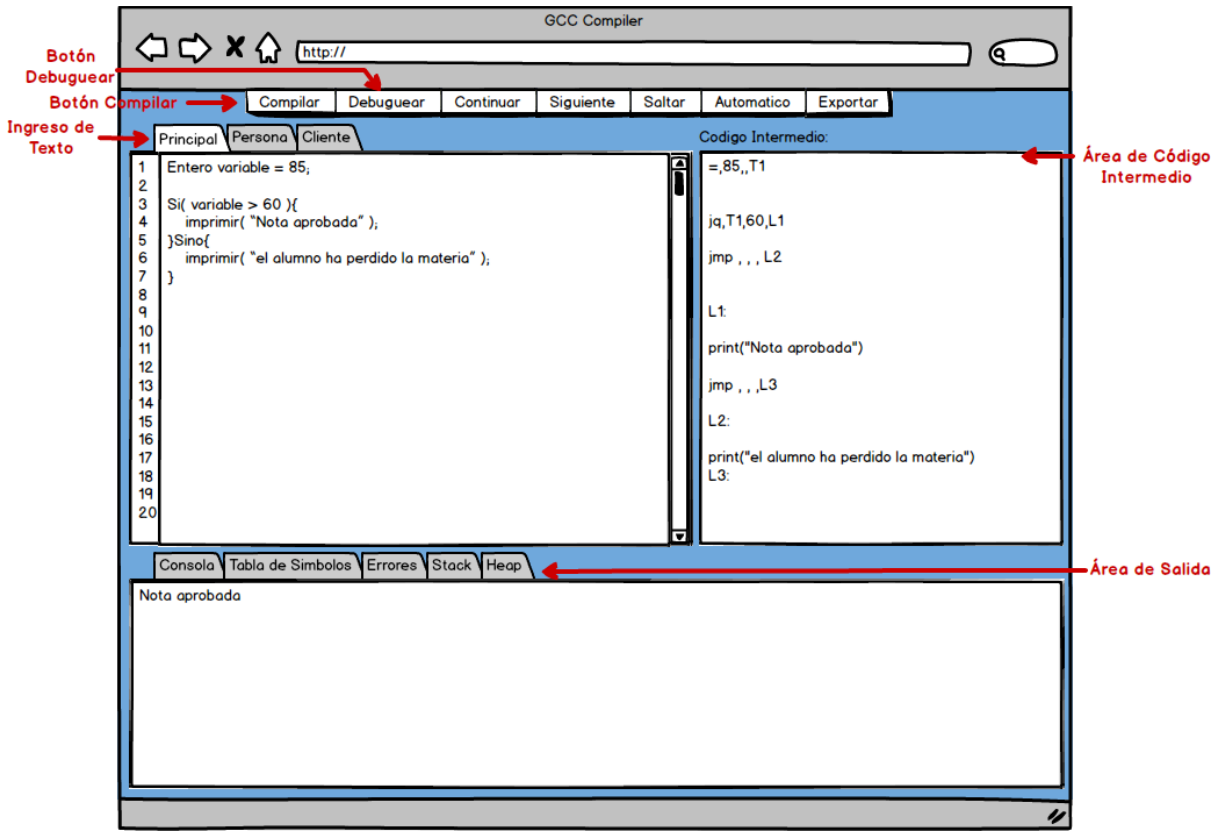
Imagen 9: pantalla para participar en una lección.



5. Si el usuario desea compilar el ejemplo entonces podrá hacer clic en el botón “compilar ejemplo” que redireccionará 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

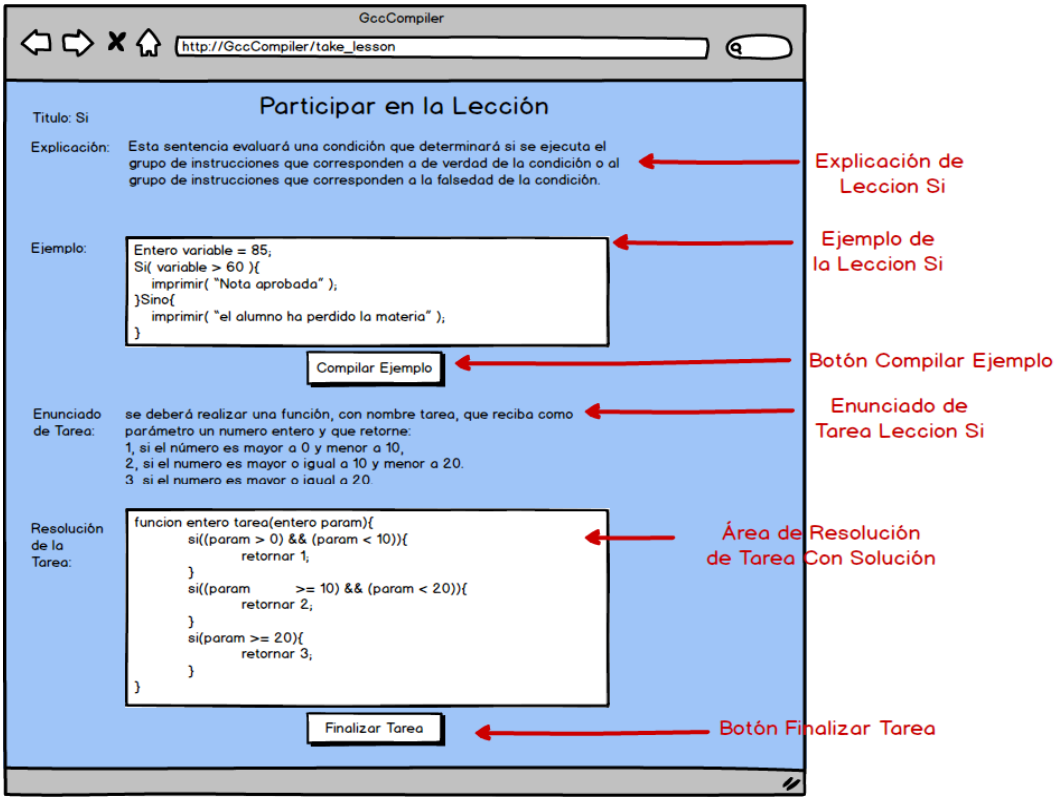
mostrará al usuario el código intermedio generado en el área de código intermedio y las salidas del lenguaje en el área de salidas. Esto se puede visualizar en la imagen 10.

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



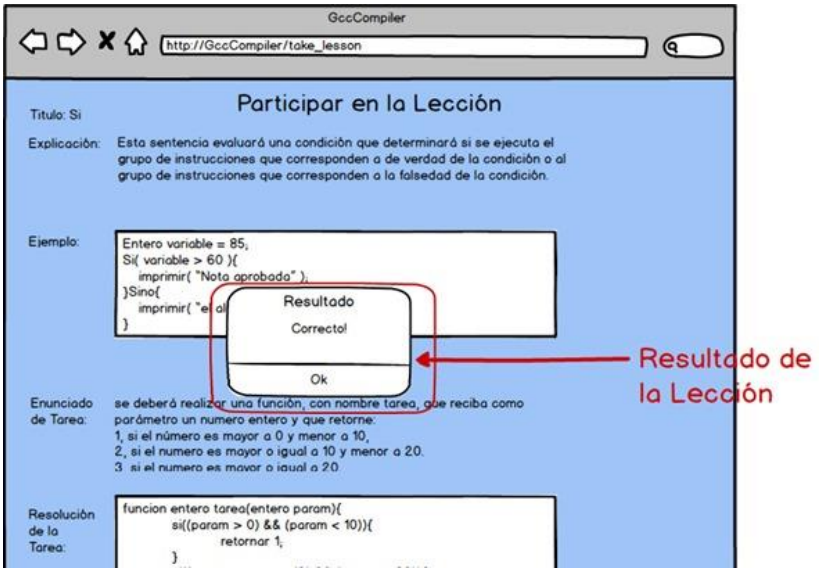
6. Después de que el usuario haya entendido la explicación y el ejemplo de la lección se podrá realizar la tarea, si es que tiene una. Esta tarea será una forma en la que el usuario podrá comprobar que haya aprendido correctamente la lección. En el espacio para subir la resolución de la tarea, el usuario podrá ingresar el código de Compiler 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.

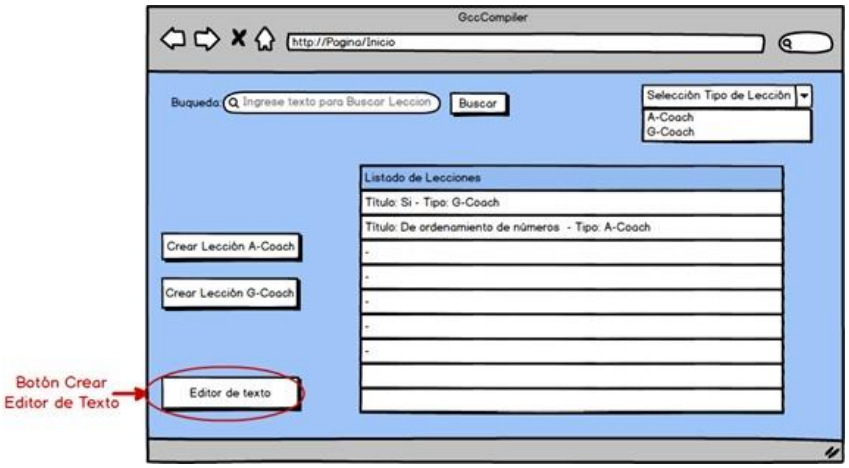


3.2.4 Ingreso de código de alto nivel libremente (fuera de una lección)

Para poder practicar lo que se ha aprendido en las lecciones se podrá ingresar código Compiler 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.

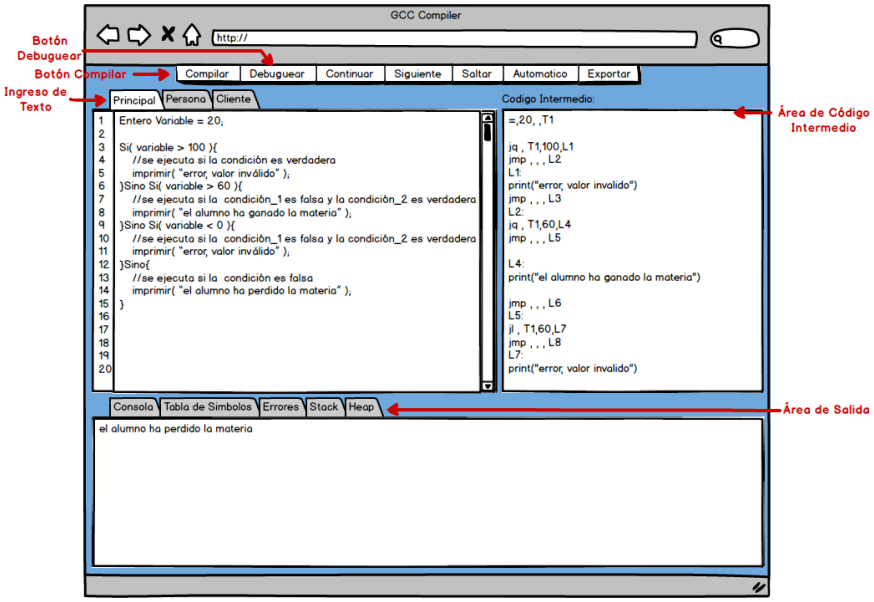
1. Hacer clic en el botón “Editor de Texto”. Esto se puede observar en la imagen 13.

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.

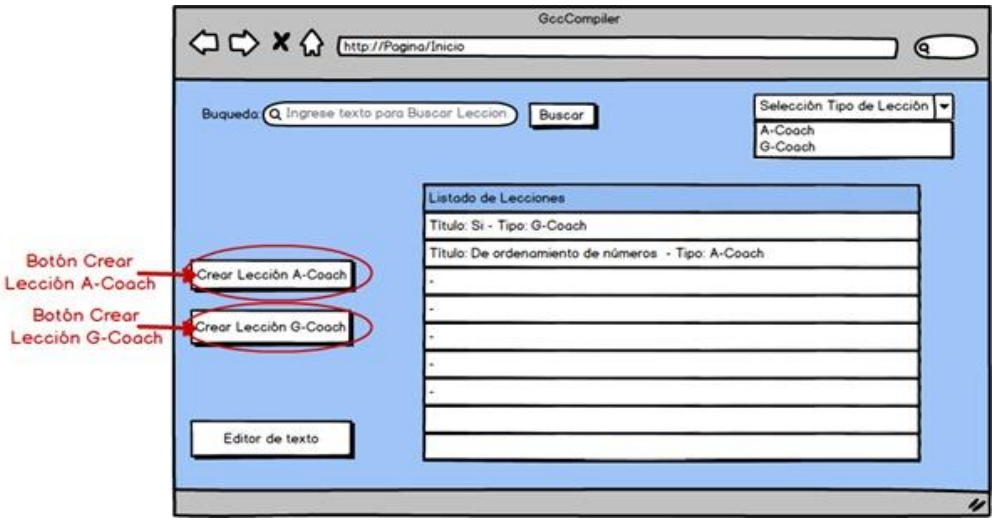


3.2.5 Carga masiva de lecciones

La carga masiva de lecciones se hará por medio de un archivo en el que se tendrán definidas una o varias lecciones de un mismo tipo que, al subirlo a la aplicación, se crearán las lecciones correspondientes. Este Para la carga masiva de lecciones el usuario deberá seguir los siguientes pasos.

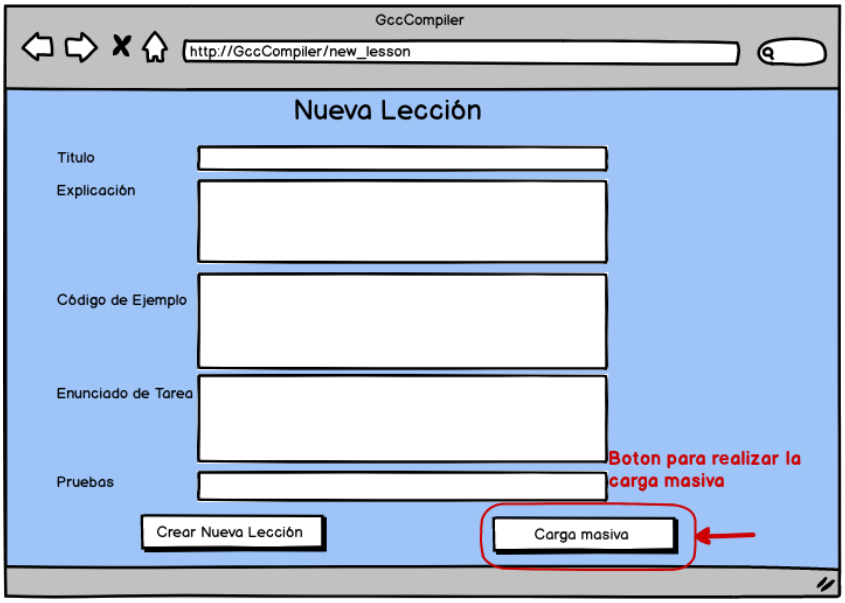
1. Ingresar a la aplicación. La aplicación mostrará la pantalla de inicio de la aplicación y se deberá hacer clic en el botón “Crear lección A-coach” o en el botón “Crear lección G-coach” como se muestra en la imagen 15.

Imagen 15: pantalla de inicio de la aplicación para cargar lecciones de forma masiva.



2. La aplicación mostrará la pantalla para el ingreso de una nueva lección. En esta pantalla el usuario dará clic sobre el botón carga masiva. Esto lo puede visualizar en la imagen 16.

Imagen 16: pantalla para crear una nueva lección.



3. La aplicación mostrará una ventana de dialogo en el que el usuario deberá seleccionar el archivo que contendrá las lecciones.

Imagen 17: pantalla para seleccionar archivo de carga masiva.

Sintaxis

```
{%
        Titulo { <<Titulo de la lección >>}
        DescripcionE { <<Descripción de la lección >> }
        Tipo { << Tipo de Lección >>}
        Ejemplo { << Ejemplo de la lección>> }
        DescripcionT { << Descripción de la tarea >> }
        Resultado { << La llamada a la función del resultado >> }
%}
```

Al cargar este archivo el usuario deberá validar que vengan los datos necesarios para poder crear la lección antes de guardarlo.

En este archivo podrá venir más de una lección, de manera que la aplicación pueda realizar una carga masiva de lecciones.

Sintaxis

```
{%
        Resultado { << La llamada a la función del resultado >> }
        Título {<<Título de la lección >>}
        Ejemplo { << Ejemplo de la lección>> }
        Tarea { << Descripción de la tarea >> }
        Descripción { <<Descripción de la lección >> }
%}
{%
        Título {<<Título de la lección >>}
        Descripción {<<Descripción de la lección >> }
```

```
        Ejemplo { << Ejemplo de la lección>> }
%}
```

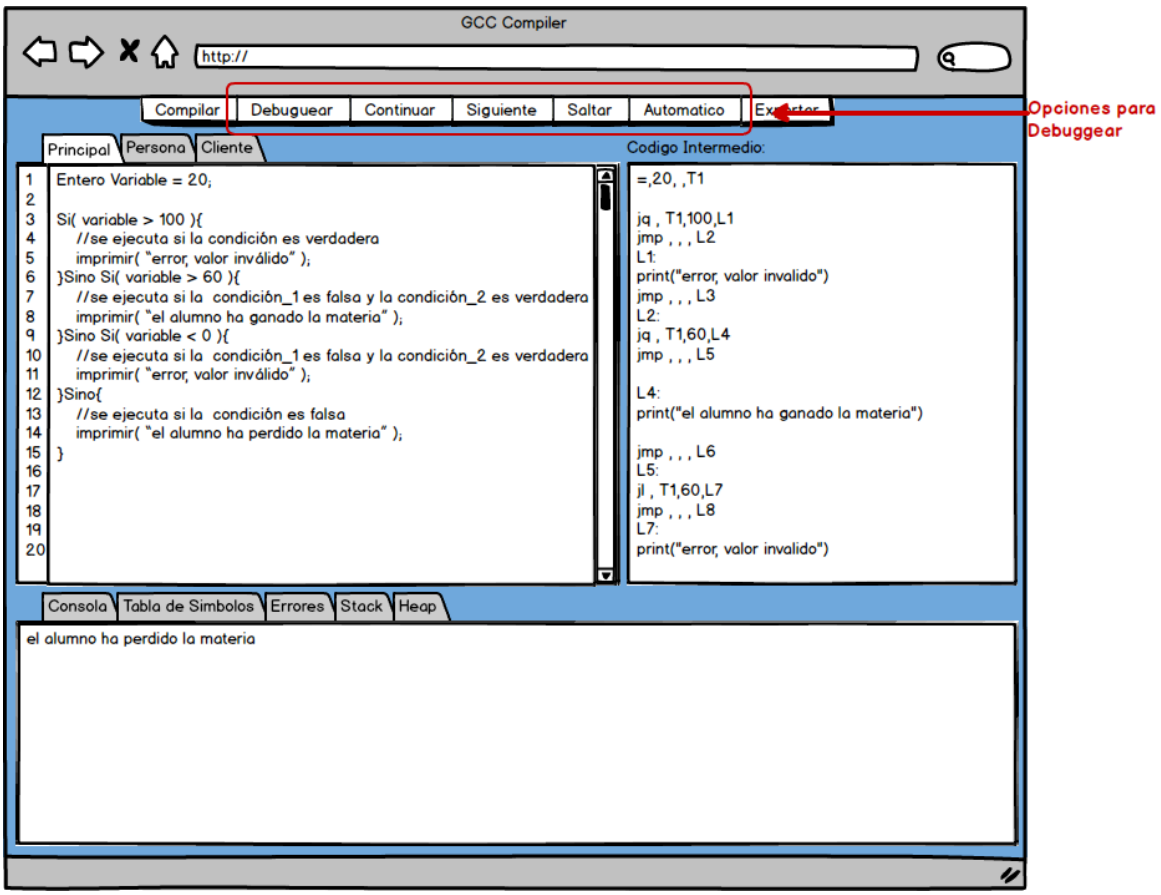
4. Al finalizar la carga de las lecciones la aplicación deberá redirigir a la página de inicio y mostrarle al usuario las lecciones cargadas en el listado de lecciones.

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.

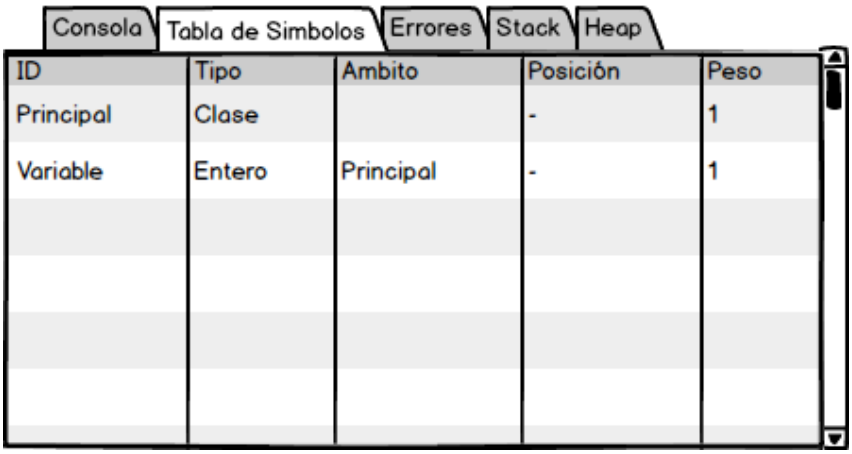
Imagen 18: sugerencia de pantalla debugger.



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.

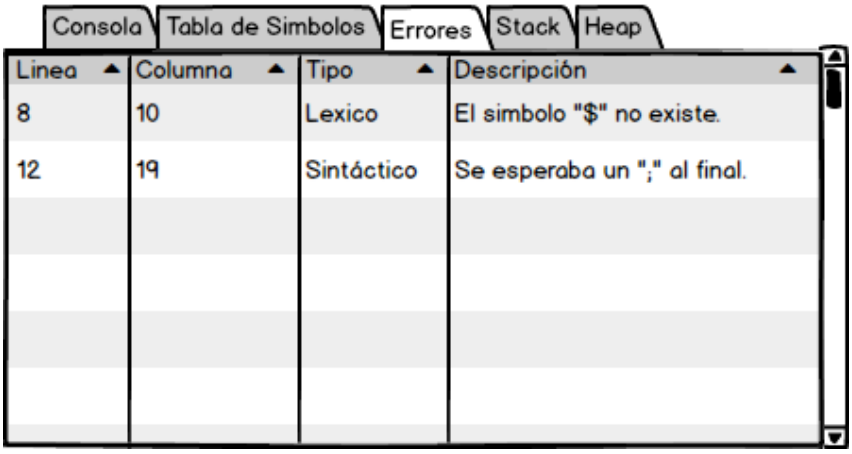
Imagen 19: sugerencia de pestaña tabla de símbolos.



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.

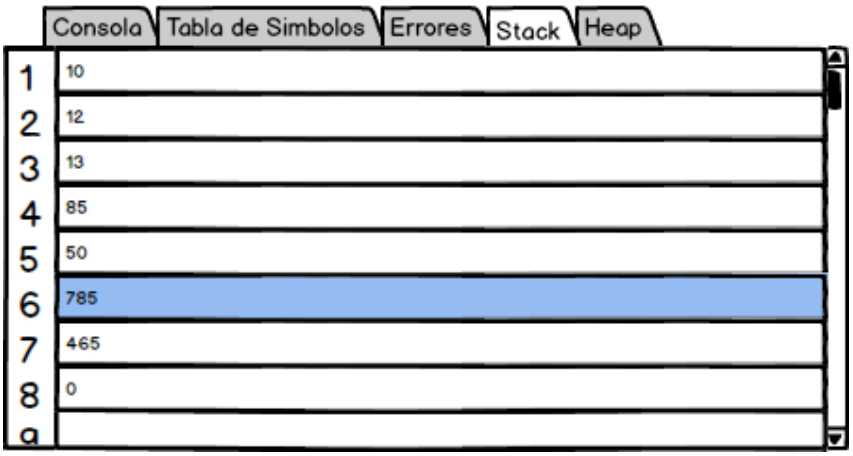
Imagen 20: sugerencia de pestaña errores.



Stack:

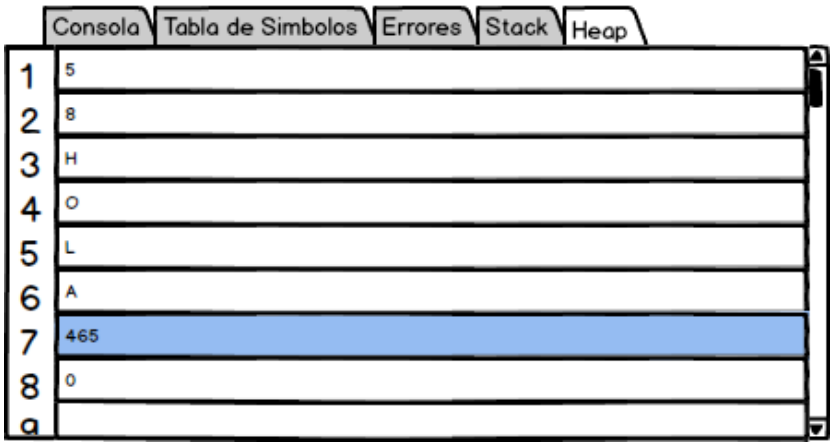
En la pestaña Stack podrá observarse la información que se encuentre hasta el instante de ejecución que se encuentre el programa que se está debugeando. También deberá indicar la posición actual del puntero del stack, la funcionalidad de esta pestaña se puede observar en la imagen 21.

Imagen 21: sugerencia de pestaña stack.



Heap: En la pestaña Heap 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, la funcionalidad de esta pestaña se puede observar en la imagen 22.

Imagen 22: sugerencia de pestaña heap.

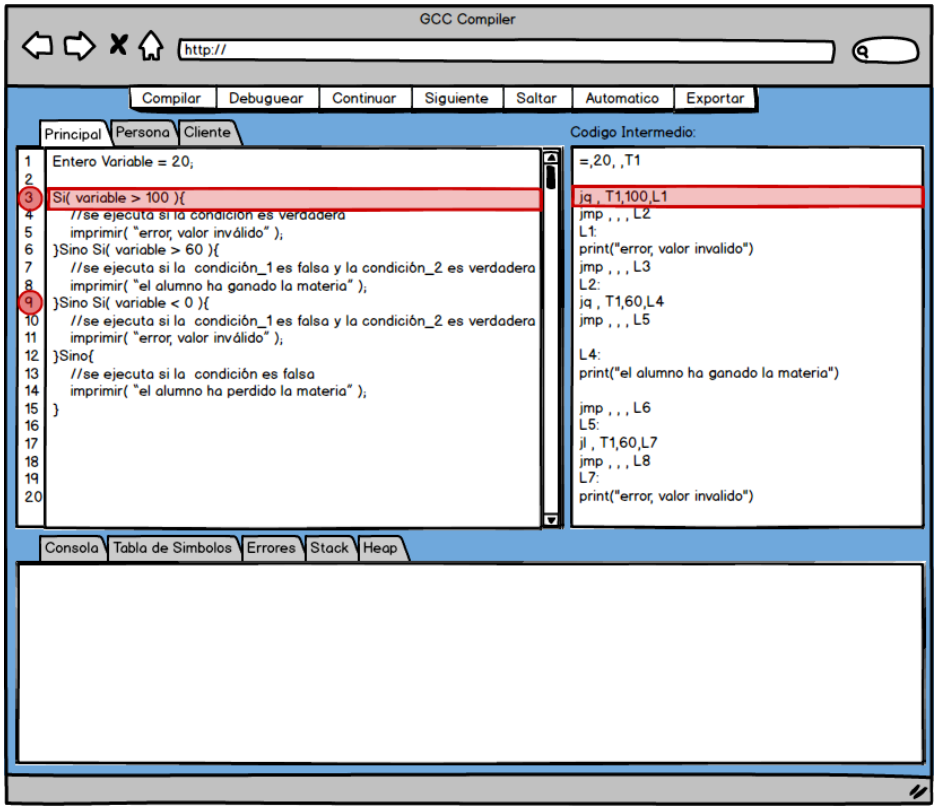


Punto de interrupción

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 23: 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

GccCompiler será capaz de manejar y reportar los errores durante el proceso de traducción de código alto nivel (compiler) a representación intermedia de cuádruplos.

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 Compiler

El lenguaje de alto nivel Compiler será un lenguaje orientado a objetos y estructurado 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.

Los tipos de datos para cada expresión vendrán dados por los sistemas de tipos definidos para las operaciones y del tipo de datos asociados al resto de elementos del lenguaje.

4.2 Características del lenguaje

Esta sección describirá las características que contiene el lenguaje Compiler.

4.2.1 Insensitive

El lenguaje es case insensitive, esto significa que no existirá diferencia entre mayúsculas y minúsculas, esto aplicará tanto para palabras reservadas propias del lenguaje como para identificadores. Por ejemplo, si se declara un identificador de nombre “Contador” este no será distinto a definir un identificador “contador”.

4.2.2 Polimorfismo

Los procedimientos y funciones deberán tener la propiedad de polimorfismo. Esto quiere decir que podrá haber una o varias funciones o procedimientos con el mismo nombre, pero que tengan diferente tipo de retorno o diferentes parámetros.

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`	true, 1 false, 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 Compiler

En este punto se definirá la sintaxis del lenguaje compiler 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- booleano decimal - decimal	decimal	5 - 4.5 = 0.5 7.8 - 3 = 4.8 15.3 - 'a' = 81.7 'b' - 2.7 = 95.3 true - 1.2 = 0.2 4.5 - false = 4.5 3.56 - 2.3 = 1.26
Entero- caracter caracter - Entero booleano - Entero Entero - booleano	entero	7 - 'c' = 92 'C' - 7 = 60 4 - true = 3 4 - false = 4

Entero - Entero		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 true * 1.2 = 1.2 4.5 * false = 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 * true = 4 4 * false= 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 true / 1.2 = 0.8333 4.5 / false = error 3.56 / 2.3 = 1.5478 7 / 'c' = 0.7070 'C' / 7 = 9.5714 4 / true = 4.0 4 / false = 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 true ^ 1.2 = 1.0 4.5 ^ false = 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 ^ true = 4 4 ^ false = 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 10 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 = true 4.5 > 6 = false "abc" > "abc" = false 'a' > "a" = false 97 > 'a' = false
<	5 < 4 = false 4.5 < 6 = true "abc" < "abc" = false 'a' < "a" = false 97 < 'a' = false
>=	5 >= 4 = true 4.5 >= 6 = false "abc" >= "abc" = true 'a' >= "a" = true 97 >= 'a' = true
<=	5 <= 4 = false 4.5 <= 6 = true "abc" <= "abc" = true 'a' <= "a" = true 97 <= 'a' = true
==	5 == 4 = false 4.5 == 6 = false "abc" == "abc" = true 'a' == "a" = true 97 == 'a' = true
!=	5 != 4 = true 4.5 != 6 = true "abc" != "abc" = false 'a' != "a" = false 97 != 'a' = false

4.3.4 Operaciones lógicas

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

Precedencia de operadores lógicos

Para saber el orden jerárquico que tienen las operaciones lógicas se define la precedencia de operadores, se considerará la precedencia de menor a mayor según el orden de aparición del operador lógico en la tabla 11.

Tabla 11: precedencia de operadores lógicos.

Operador lógico
OR
XOR
AND
NOT

En la tabla 12 se muestra la tabla de verdad de cada operador lógico.

Tabla 12: operadores lógicos

<table>
<tr><th>Operación lógica</th><th>Operador</th><th>Tabla de verdad</th></tr>
<tr><td>OR</td><td>||</td><td>
<table>
<tr><th>Op1</th><th>Op2</th><th>Op1 || Op2</th></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>XOR</td><td>??</td><td>
<table>
<tr><th>Op1</th><th>Op2</th><th>Op1 ?? Op2</th></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>AND</td><td>&&</td><td>
<table>
<tr><th>Op1</th><th>Op2</th><th>Op1 && Op2</th></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td></td><td>
<table>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>NOT</td><td>!</td><td>
<table>
<tr><th>Op1</th><th>! Op1</th></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td></tr>
</table>
</td></tr>
</table>

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. Las variables que se declaren afuera de los métodos serán variables globales.

Sintaxis

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

Ejemplo

```
Entero Identificador;
//si se desea inicializar una variable
Entero numero = 10;
```

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 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 ] )*;
```

Ejemplo

```
Decimal arreglo[4][3];
caracter cadena[10];
```

4.3.8 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;

caracter cadena[10];
//Esta asignación solamente será válida con los arreglos de caracteres.
cadena = “mi cadena”;
```

4.3.9 Tamaño de un 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
```

4.3.10 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,Expresion);
```

Ejemplo

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

cadena = “hola”
cadena2=” mundo”

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

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 = true;
```

```
Decimal miDecimal = 0.1;

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

4.3.11 Casteo

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 = true;
Decimal miDecimal = 1.25;

cadena = “5”;

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

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

4.3.12 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’);
```

4.3.13 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;
    }
}
```

4.3.14 Visibilidad

La visibilidad se refiere 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 tipos de visibilidad aceptados por el lenguaje serán:

- Público: permitirá acceder a los miembros de una clase desde cualquier lugar. La palabra reservada para esta visibilidad 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 para esta visibilidad será “protegido”.
- Privado: permitirá acceder a los miembros de una clase sólo desde la misma clase. La palabra reservada para esta visibilidad será “privado”.

Notas:

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

4.3.15 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
}
```

4.3.16 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.

El envío de los parámetros varía según sea el tipo; entre parámetro por referencia, que indica que cualquier modificación sobre dicho parámetro, dentro del cuerpo del método que fue llamado, es como que se estuviera modificando la misma estructura o variable que se ha enviado: Los cambios realizados en el parámetro se verán reflejados en el elemento original; o bien se pueden enviar parámetros por valor, lo que significa que los cambios realizados sobre el parámetro NO tienen efecto sobre el valor original del elemento. A continuación, se listan las distintas formas de enviar parámetros.

- Los tipos numéricos o booleanos siempre serán enviados por valor.
- El tipo cadena será enviado, por defecto, por valor; a no ser que se indique explícitamente que se desea enviar por referencia, esto se hará por medio del método obtenerDireccion.
- Para recibir como parámetro algún valor por referencia, se podrá el tipo, la palabra reservada puntero y el id del parámetro.
- Las estructuras será posibles enviarlas por valor o por referencia.
- Las clases siempre serán enviadas por referencias.

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 parametro2) {
    retornar operador1 + operador2;
}

entero suma(Persona puntero per, entero parametro2) {
    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;
}
```

4.3.17 Retornos

Esta sentencia puede venir a cualquier nivel de profundidad dentro del código e interrumpirá toda la ejecución del método o función que esté corriendo en ese momento. El retorno vacío es exclusivo para los métodos, mientras que el retorno con una expresión es para controlar las funciones.

Como se mencionaba anteriormente, el retorno será exclusivo para métodos y funciones, pero este se encontrará en cualquier parte del código y podrá encontrarse más de uno en un solo método o función.

Sintaxis

```
//Retorno para un método
retorno;

//Retorno para una función
retorno elemento;
```

4.3.18 Llamada a procedimientos y funciones

Para llamar un procedimiento o función en el lenguaje Compiler 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 clase 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;
    }
}
```

4.3.21 Sentencias

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”);
```

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();
```

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);
```

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”;
carlos.setEdad(22);
```

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
Carácter var2[15] = concatenar( concatenar( “cadena ”, “a”),“ concatenar”);
```

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;
```

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,** también se podrá asignar un nulo de la siguiente manera **{‘\0’}** o **{“\0”}.**

Ejemplo

```
//asignación de un nulo con la palabra reservada Nada.
Caracter persona[20] = Nada;
```

```
Caracter letra = Nada;

//asignación de un nulo con el valor {'\0'}
Caracter persona_2[10] = {'\0'};
Caracter letra_2 = {'\0'};
```

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}}
```

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;
```

4.3.22 Sentencias rompe Ciclos

Instrucción romper

Esta instrucción tiene la funcionalidad de romper o interrumpir la ejecución de otra instrucción, si esta se encuentra dentro de un ciclo hará que la ejecución del mismo finalice y da el foco de la ejecución a la siguiente instrucción fuera del ciclo.

Esta sentencia deberá de ser utilizada solo por las sentencias de flujo.

Sintaxis

```
romper;
```

Instrucción continuar

Esta sentencia se aplicará únicamente a las sentencias de flujo, su función es interrumpir la iteración actual para saltarse el resto de sentencias y empezar la siguiente iteración.

Sintaxis

```
continuar;
```

4.3.23 Estructuras

Estructuras

Las estructuras son colecciones de variables relacionadas bajo un nombre. Las estructuras pueden contener variables de muchos tipos diferentes de datos.

La palabra reservada **Estructura** estará definiendo una estructura. Está llevará un nombre que será el identificador de la estructura. Las variables declaradas dentro de los corchetes de la definición de estructura serán los miembros de la estructura. Los miembros de la misma estructura deben tener nombres únicos. Cada definición de estructura debe terminar con un punto y coma. Todas las estructuras deberán de contar con al menos una propiedad

Sintaxis

```
Estructura <id> [
        //declaraciones
];
```

Ejemplo

```
Estructura Persona [
        Caracter nombre[25];
        Caracter genero[20] = Nada; // M -> masculino, F -> Femenino
        Entero edad;
        Entero posición = 0;
        //se agrega que apunta a otra estructura
        Persona amigo;
];
```

Inicializar estructura

Una estructura reservara espacio de memoria al momento de ser declarada. También se podrán declarar punteros a estructuras esto se explicará en la siguiente sección.

Ejemplo

```
//Se reserva el espacio de memoria para la estructura persona
Persona usuario;
//Inicialización de la estructura usuario
Usuario.nombre = "Daniel";
```

```
Usuario.genero = 'm';
Usuario.edad = 23;
Usuario.posicion = 0;
```

Asignación de un valor a un elemento de una estructura.

Se podrá asignar o cambiar un valor a una variable dentro de una estructura. La asignación se hará utilizando el nombre de la variable con la que se inicializo la estructura seguido por un punto y el nombre de la variable que se cambiará el valor.

Si la estructura no fue inicializada no se podrá asignar un valor a la variable.

Ejemplo

```
Persona usuario_2;
//cambio de valor de una variable dentro de la estructura
usuario.nombre = “Maria Fernanda”;
usuario.edad = 25;
usuario.amigo = usuario_2;

//error para cambia variable sin inicialización
usuario_4.nombre = “Mario”; //esto es un error, no fue inicializado usuario_4
```

4.3.24 Punteros

Punteros

Un puntero será un tipo especial de variable que almacena una dirección de memoria, este podrá ser de cualquier tipo primitivo sea entero, caracter, booleano, decimal o podrá apuntar a una dirección de memoria en donde se encontrará almacenada una estructura. Para crear un puntero se utilizará el método crearPuntero que declarará la variable como un puntero del tipo de variable que se especifique.

Para obtener la dirección de memoria de una variable se utilizará el método obtenerDireccion.

Sintaxis

```
crearPuntero(Tipo, <id1>);//declaración de un puntero

crearPuntero(Tipo, <id2>) = <id1> ;//declaración del puntero id2 que apunta a la misma dirección de memoria que el puntero id1

<id1> = <id2>;//id1 apunta a la misma dirección de memoria que id2

Tipo <id3> ;//declaración de una variable

<id1> = obtenerDireccion(<id3>);//id1 apunta a la dirección de memoria en donde se encuentra almacenada la variable id3.
```

Ejemplo

```
Entero ubicación = 0;//ubicación guarda el valor de 0.
Caracter cad = "hola mundo";

crearPuntero(Entero, dirección) = obtenerDireccion(ubicación); //dirección almacenará la posición de memoria de ubicación.

crearPuntero(Caracter, c) = obtenerDireccion(cad);  //cadena apuntará a la dirección de memoria del caracter 'h'
```

reservarMemoria

Función propia del lenguaje Compiler que recibirá un valor numérico que será el espacio que deberá reservar en heap para determinada estructura y retornará un puntero al inicio de la estructura

Ejemplo

```
crearPuntero(Persona, a) = reservarMemoria (20);
```

consultarTamanio

Función propia del lenguaje Compiler que recibirá una expresión o variable y deberá retornar el valor numérico del tamaño que ocupa en heap.

Ejemplo

```
Entero size = consultarTamanio (estructura); //puntero de tipo persona que apunta a nulo.
```

Creación de punteros a estructuras.

Un puntero podrá direccionar a nulo o podrá direccionar a una posición de memoria en donde se encontrarán almacenados los valores de una estructura.

Ejemplo

```
crearPuntero(Persona, a) = Nada; //puntero de tipo persona que apunta a nulo.

crearPuntero(Persona, a) = reservarMemoria(conserguirTamanio(Persona));
//puntero de tipo persona que apunta a la dirección de memoria que retornará la función reservarMemoria.
```

Asignación de un valor a un elemento de una estructura tipo puntero.

Se podrá asignar o cambiar un valor a una variable dentro de una estructura de tipo puntero. La asignación se hará utilizando el nombre de la variable con la que se inicializará la estructura seguido por una flecha (->) y el nombre de la variable a la que se cambiará el valor.

Si la estructura no fuera inicializada no se podrá asignar un valor a la variable.

Ejemplo

```
//cambio de valor de una variable dentro de la estructura
crearPuntero(Persona, usuario) = reservarMemoria (conserguirTamanio
(Persona));

Usuario->nombre = “Maria Fernanda”;
usuario->edad = 25;
```

Nota: El lenguaje manejará puntero de punteros

Para quitar la referencia de un puntero se hará de la siguiente manera.

Sintaxis

```
destruirPuntero(Id1);
```

Ejemplo

```
//cadena de caracteres
crearPuntero(caracter, cad) =  "HOLA MUNDO";
//se ejecuta mientras no sea fin de cadena
Mientras (cad != '0' ){
  imprimir (cad); //imprime el caracter actual
  cad++; //pasa a la siguiente direccion de memoria.
}
destruirPuntero(cad);
```

4.3.25 Callback

Funciones pasadas como parámetros

Un callback es código ejecutable que se pasará como argumento a otro código, el cual estará esperando por un call back para ejecutar el código que ha sido pasado como parametro. En general un callback es el paso de una función como parámetro de otra función.

Ejemplo

```
//función callback
Entero FuncionLlamada(entero param) {
    imprimir (concatenar(“parámetro: ”, convertirAcadena( param ));
    retornar param;
}

Entero FuncionLlamada2(funcion param) {
    param = param + 500;
    imprimir (concatenar(“parámetro: ”, convertirAcadena( param ));
    retornar param;
}

vacio Llamador(funcion funcionParametro ){
```

```
    Entero val =funcionParametro(1995); //llamada desde una función como parámetro
}

Principal(){
    Llamador(obtenerDireccion(FuncionLlamada)); //la salida de esta llamada seria 1995
    Llamador(obtenerDireccion(FuncionLlamada2)); //la salida de esta llamada seria 2495
}
```

4.3.26 Estructuras de Datos

El lenguaje Compiler, 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.

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();
```

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->

miIndice = miLista.buscar(50); //miIndice=1
```

Pila

La pila es una estructura de datos que agrega los elementos que al inicio de la estructura. En Compiler 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->
```

Cola

La cola es una estructura de datos que agrega los elementos al final de la estructura. En Compiler 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->
```

4.3.27 Sentencias de Control

Si

Esta sentencia evaluará una condición que determinará qué grupo de sentencias va a ejecutarse.

Sintaxis

```
Si ( condición_1 ) {
   Es_verdadero{
      //se ejecuta si la condición es verdadera
      //varias sentencias
   }
   Es_falso( condición_2 ){
      //se ejecuta si la  condición_1 es falsa y la condición_2 es verdadera
      //varias sentencias
   }
Fin-si
```

Ejemplo

```
Si( variable > 100 || variable < 0 ) {
   Es_verdadero {
      imprimir( “error, valor inválido” );
   }
   Es_falso{
      Si( variable > 60 ){
         Es_verdadero {
            imprimir( “el alumno ha ganado la materia” );
         }
         Es_falso {
            imprimir( “el alumno ha perdido la materia” );
         }
      Fin-si
   }
Fin-si
```

Evaluar_si

Esta instrucción permitirá realizar varias comparaciones, cada comparación será un caso. La instrucción deberá ejecutar las instrucciones correspondientes a cada valor que sea igual al valor que se estará evaluando.

Los valores que deberá de valuar en cada caso puede ser cualquier tipo de dato.

Sintaxis selección

```
Evaluar_si (<expresión>) {
        //listado de casos.
}
```

Sintaxis caso

```
Es_igual_a <valor>:
        //lista de sentencias
```

Ejemplo

```
Evaluar_si (var){
        Es_igual_a 1:
                //instrucciones
                romper;
        Es_igual_a 'a':
                //instrucciones
                romper;
        Es_igual_a "casa":
                //instrucciones
                romper;
        defecto:
                //instrucciones
}

Evaluar_si (var2){
        Es_igual_a 100:
                //instrucciones
                romper;
        Es_igual_a 'b':
                //instrucciones
                romper;
        Es_igual_a "elemento":
                //instrucciones
                romper;
        defecto:
                //instrucciones
}
```

4.3.28 Sentencias de Flujo

`Repetir_Mientras`

Esta sentencia repetirá un conjunto de instrucciones mientras la condición sea verdadera.

Sintaxis

```
Repetir_Mientras( condición ) {
    //se repetirán si la condición es verdadera
    //varias sentencias
}
```

Ejemplo

```
Repetir_Mientras( Persona.estado != 2){
   imprimir (concatenar("Estado :", convertirAcadena(Persona.estado));
   actualizar_estado(Persona);
}
```

Hacer-Mientras

Esta sentencia ejecutará un grupo de instrucciones y las seguirá ejecutando mientras la condición sea verdadera.

Sintaxis

```
hacer {
   //se ejecuta la primera vez y se seguirá repitiendo si la condición es verdadera
   //varias sentencias
}mientras ( condición );
```

Ejemplo

```
hacer{
   b++;
   a = b + c;
}mientras( a < 100);
```

ciclo-X

Esta sentencia ejecutará un grupo de instrucciones si una de las dos condiciones es verdadera y las seguirá ejecutando mientras ambas condiciones sean verdaderas.

Sintaxis

```
Ciclo_doble_condicion ( condición_1 , condición_2 ) {
   //varias sentencias
}
```

Ejemplo

```
Ciclo_doble_condicion ( bandera == true , contador < 10 ) {
  imprimir("estoy dentro del ciclo-x");
  contador++;
```

```
}
```

Repetir

Esta sentencia repetirá un conjunto de instrucciones hasta que la condición sea verdadera.

Sintaxis

```
Repetir{
    //se repite hasta que la condición sea verdadera
   //varias sentencias
}hasta_que ( condición );
```

Ejemplo

```
Repetir{
    imprimir (concatenar("Posición :", convertirAcadena(Persona.posicionX));
    Persona.mover();
}hasta_que( Persona.posicionX > 100);
```

Repetir_contando

Es un ciclo que permitirá declarar una variable que servirá para llevar el conteo de las repeticiones del ciclo. Se deberá ingresar un valor de inicio en el campo "desde" y un valor de final en el campo "hasta". Este ciclo deberá seguir el siguiente comportamiento:

- Si el valor desde es menor al valor hasta entonces la variable irá aumentando en 1 cada iteración.
- Si el valor desde es mayor al valor hasta entonces la variable irá decrementando en 1 cada iteración.
- Si el valor desde es igual al valor hasta no se deberá entrar al ciclo.

Sintaxis

```
Repetir_contando (variable: id; desde: 0; hasta: 10)
    //se repite hasta que se llegue al valor final
    //varias sentencias
}
```

Ejemplo

```
Repetir_contando (variable: contador; desde: 0; hasta: 10)
    imprimir (convertirAcadena(contador));
    // salida: 0,1,2,3,4,5,6,7,8,9
}
```

Ciclo enciclar

Sentencia enciclo que llevara asociado un identificador, este ciclo ejecutará sus instrucciones infinitamente hasta que encuentre una sentencia romper, que le indicará cuando debe finalizar. Este ciclo deberá llevar obligatoriamente la instrucción romper.

Sintaxis

```
Enciclar <id> {
        //instrucciones dentro del ciclo
        romper;
}
```

Ejemplo

```
Enciclar ciclo_1 {
        Enciclar ciclo_2 {
                romper;
        }
        romper;
}
```

Ciclo Contador

Sentencia de ciclo que ejecutará instrucciones las veces que se le indiquen según un valor numérico. Este ciclo es similar al ciclo mientras, con la diferencia que la expresión esperada no es una condicional, sino una expresión de valor numérico tipo entero.

Sintaxis

```
Contador (<expresión>) {
        //instrucciones dentro del flujo
}
```

Ejemplo

```
Contador (5+5*5){
        Contador (10){
                //instrucciones dentro del flujo
        }
}
```

4.3.29 Entrada o Lectura de Datos

En compiler 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 True o false”, Id_4);
Leer_Teclado ( “Ingrese un número decimal”, Id_5);
```

4.4 Generación de código intermedio

Cuando el compilador termine la fase de análisis de un programa escrito en GccCompiler, realizará una transformación a una representación intermedia equivalente al código de alto nivel del lenguaje mencionado anteriormente. La representación intermedia para el lenguaje Compiler será una representación intermedia de cuádruplos.

El formato de cuádruplos es una representación reconocida por el compilador, a diferencia del código 3 direcciones el formato de cuádruplos tiene 4 campos llamados: operador, argumento1, argumento2 y resultado. El campo operador contiene un código interno para el operador (+,*,-..ect.). Por ejemplo, la instrucción de tres direcciones x = y + z se representa colocando a "+" en el campo operador, "y" en el campo argumento1, "z" en el campo argumento2 y "x" en el campo de resultado.

Ejemplo

Suponiendo: x = y + z

Operador	Argumento 1	Argumento 2	Resultado
+	y	z	x

En la generación de cuádruplos se deberá de represar de la siguiente manera y delimitados por ","

Sintaxis

```
Operador, argumento_1, argumento_2, resultado
```

Ejemplo

```
+, t1, 5, t2
*, t2, 10, t3
```

4.4.1 Temporales

Los temporales serán creados por el compilador en el proceso de generación de código intermedio. Los temporales deberán empezar con la letra "t" seguida de un número.

Ejemplo

```
=, 0, , t1
+, t5, t1, t6
```

4.4.2 Etiquetas

Las etiquetas serán creadas por el compilador en el proceso de generación de código intermedio. Las etiquetas deberán empezar con la letra "L" seguida de un número.

Ejemplo

```
L1:
   =, 0, , t1
   +, t5, t1, t6
```

4.4.3 Operadores aritméticos

Las operaciones aritméticas contarán con argumeton1, argumeto2, resultado y un operador perteneciente a la siguiente tabla 14.

Tabla 14: Operaciones aritméticas en representación intermedia de cuádruplos.

Operación	Símbolo	Cuádruplos
Suma	+	+, t2, 6, t3
Resta	-	-, t2, 6, t3
Multiplicación	*	*, t2, 6, t3
División	/	/, t2, 6, t3

4.4.4 Saltos condicionales

Los saltos condicionales contarán con arg1, arg2, resultado y op. Resultado contendrá la etiqueta destino y el operador pertenecerá en la tabla 15.

Tabla 15: Operaciones relacionales en código de tres direcciones

Operación	Símbolo	Equivalente	Ejemplo
Igual que	je	==	je, t1, t2, L1
Diferente que	jne	!=	jne, t1, t2, L1
Mayor que	jg	>	jg, t1, t2, L1
Menor que	jge	>=	jge, t1, t2, L1
Mayor o igual que	jl	<	jl, t1, t2, L1
Menor o igual que	jle	<=	jle, t1, t2, L1

Ejemplo

```
L1:
   +, 0, 5, t1
```

```
    *, t1, t2, t3
   =, 5, , t1
   jl, t1, t2, L1
    *, t1, t2, t3
   =, 5, , t1
L2:
```

4.4.5 Salto Incondicional

Los saltos incondicionales contarán con resultado y op. Resultado contendrá la etiqueta destino. Este realizará el salto hacia una etiqueta que se le especifique.

Sintaxis

```
jmp, , , etiqueta
```

Ejemplo

```
L1:
   +, t2, 6, t3
   jmp, , , L2
L2:
```

4.4.6 Asignación

La asignación podrá tener cualquiera de las siguientes formas.

Sintaxis

```
operador, argumento, argumento, identificador
=, argumento, , identificador
```

Ejemplo

```
+, 0, 5, t1
*, t1, t2, t3
=, 5, , t1
```

4.4.7 Declaración de Métodos

Los métodos podrán ser declarados de la siguiente forma.

Sintaxis

```
begin, , , metodo1
.
.
.
end, , metodo1
```

Ejemplo

```
begin, , , metodo1
   +, 0, 5, t1
   *, t1, t2, t3
   =, 5, , t1
end, , metodo1
```

4.4.8 Llamada a Métodos

La llamada a métodos se realizará de la siguiente forma.

Sintaxis

```
call, , , metodo1
```

Ejemplo

```
begin, , , metodo1
+, 0, 5, t1
*, t1, t2, t3
=, 5, , t1
call, , , metodo2
end, , metodo1
```

4.4.9 Acceso a arreglos

El sistema de representación intermedia de cuádruplos contará con dos arreglos, el stack que tendrá la función de la pila vista en clase y el heap visto en el laboratorio.

Sintaxis

```
<=, t1, t2, stack
=>, t1, t2, stack
```

Ejemplo

Ejemplo de cuádruplos	Equivalente
<=, t1, t2, stack	stack[t1] = t2
=>, t1, t2, stack	t2 = stack[t1]

4.4.10 Print

Esta sentencia recibirá un parámetro de tipo cadena y un identificador que tendrá el valor de lo que se mostrará en la consola de salida. En la tabla 16 se muestran los parámetros permitidos para esta sentencia.

Sintaxis

```
print( “parámetro”, identificador );
```

Tabla 16: Parámetros del método print

Parametro	Acción
“%c”	Imprime el valor caracter del identificador.
“%d”	Imprime el valor entero del identificador.
“%f”	Imprime el valor con punto flotante del identificador.

Ejemplo

```
print(“%d” , t1);
```

4.4.11 Entrada o lectura de dato

Esta es una función nativa de la representación intermedia de cuádruplos, esta nos permitirá interrumpir el flujo normal de la ejecución para pedir un valor al usuario. Esta recibirá los parámetros de las posiciones en la pila descritos en la tabla 17.

Sintaxis

```
call, , , $_in_value
```

Tabla 17: Parámetros del método $_invalue en representación intermedia de cuádruplos

Posición de la pila	Descripción
0	Retorno, no es usado
1	Referencia donde guardará el valor.
2	Referencia a la cadena que mostrará

Ejemplo

```
begin, , , metodo1
```

```
+, 0, 5, t1
*, t1, t2, t3
=, 5, , t1
call, , , $_in_value
end, , metodo1
```

5 Entregables y Restricciones

5.1 Entregables

- Código fuente
- Aplicación Funcional
- Gramáticas
- Manual Técnico (como mínimo diagrama de flujo de GccCompiler, herramientas utilizadas y gramática del lenguaje).
- Manual de Usuario (mínimo el funcionamiento de la página principal, funcionamiento del traductor y el debugger).

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 forma de entrega será virtual, los detalles de la entrega se informarán días antes de la fecha de entrega. La calificación se realizará sobre archivos ejecutables. Se proporcionarán archivos de entrada al momento de calificación.

5.2 Restricciones

- La aplicación deberá ser desarrollada utilizando el lenguaje javascript con node js.
- La generación de código intermedio se deberá utilizar la herramienta jison.
- El intérprete de la representación intermedia de cuádruplos deberá de ser desarrollado utilizando la herramienta jison.

5.3 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:

- GccCompiler
- Lecciones
- Tipos de lecciones
    - G-Coach
- Pantallas de la Plataforma
    - Pantalla inicio
    - Editor de texto
    - Participar en lección
    - Nueva lección
    - Evaluar tareas
- Compiler
    - Comentarios
    - Operaciones Aritméticas

    - Operadores Relacionales
    - Operaciones lógicas
    - Declaración de variables
    - Asignación de variables
    - Declaración de arreglos
    - Asignación a posiciones dentro del arreglo
    - Tamaño de un vector
    - Operaciones con cadenas
    - Casteo
    - Imprimir
    - Visibilidad
    - Herencia
    - Procedimientos
    - Funciones y procedimientos
    - Llamada a procedimientos y funciones
    - Procedimiento Principal
    - Retornos
    - Constructor
    - Sentencias
    - Sentencias rompe Ciclos
    - Estructuras de Datos
    - Sentencias de Control
    - Sentencias de Flujo
    - Punteros

- Código intermedio
    - Temporales
    - Etiquetas
    - Operadores aritméticos
    - Operadores Relacionales
    - Asignación
    - Salto Incondicional
    - Salto Condicional
    - Salto Condicional ifFalse
    - Declaración de Métodos
    - Llamada a Métodos
    - Acceso a arreglos
    - Print