

Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ingeniería en Ciencias y Sistemas

Organización de Lenguajes y Compiladores 2

Vacaciones Primer Semestre 2023

Catedrático: Ing. Luis Espino

Tutor académico: Diego Obín



USAC
TRICENTENARIA
Universidad de San Carlos de Guatemala

PyTypeCraft

Proyecto Segunda Fase de laboratorio

Competencias	4
Competencia general	4
Competencias específicas	4
Descripción	5
Descripción General	5
Flujo específico de la aplicación	5
Proceso de Compilación	5
Proceso de Optimización	6
Proceso de Comprobación	6
Proceso de Ejecución	7
Visualización de Reportes	7
Componentes de la aplicación	8
Página de Bienvenida	8
Página de Editor	9
Página de Reportes	10
Sintaxis de PyTypeCraft	11
Generalidades	11
Sistema de tipos	11
Tipos básicos	12
Tipos compuestos	12
Reglas	12
Comprobación dinámica	13
División entre cero	13
Índice fuera de los límites	14
Expresiones	15
Aritméticas	15
Multiplicación	16
División	17
Potencia	17
Módulo	17
Relacionales	18
Lógicas	18
Impresión	19
Asignaciones	19
Funciones	20
Creación de funciones	20
Funciones Nativas	21
Llamada a funciones	21
Paso por valor o por referencia	21
Condicionales	21
Loops	22
Ciclo while	22
Ciclo for	23

Sentencias de transferencia	23
Arreglos	24
Structs	25
Generación de Código Intermedio	27
Tipos de dato	27
Temporales	27
Etiquetas	27
Comentarios	28
Saltos	28
Saltos incondicionales	28
Saltos condicionales	29
Asignación a temporales	29
Métodos	30
Llamada a métodos	30
Impresión en consola	30
Estructuras en tiempo de ejecución	31
Stack	31
Heap	32
Acceso y asignación a estructuras en tiempo de ejecución	32
Encabezado	32
Método main	33
Comprobación de código tres direcciones	33
Reportes generales	41
Reporte de Tabla de Símbolos	41
Reporte de Tabla de errores	41
Reporte de Optimización	41
Manejo de errores	42
Errores semánticos	42
Entregables y Calificación	43
Entregables	43
Restricciones	43
Consideraciones	43
Calificación	44
Entrega de proyecto	45

1. Competencias

1.1. Competencia general

Que los estudiantes apliquen los conocimientos adquiridos en el curso para la construcción de un compilador utilizando las herramientas establecidas.

1.2. Competencias específicas

- Que los estudiantes utilicen herramientas para la generación de analizadores léxicos y sintácticos.
- Que los estudiantes apliquen los conocimientos adquiridos durante la carrera y el curso para el desarrollo de la solución.
- Que los estudiantes realicen análisis semántico, la generación de código intermedio y optimización del código intermedio del lenguaje PyTypeCraft.
- Que los estudiantes generen una traducción de código de alto nivel a código de tres direcciones.

2. Descripción

2.1. Descripción General

PyTypeCraft es un lenguaje de programación basado en Typescript que tiene instrucciones limitadas con la ventaja de un lenguaje dinámico con el rendimiento de un lenguaje compilado, por lo tanto, es necesario implementar un entorno de desarrollo web para construir un compilador para asegurar un buen rendimiento.

2.2. Flujo específico de la aplicación

La aplicación se compone de tres procesos importantes que son el proceso de compilación, comprobación y ejecución. Estos procesos se describen a continuación.

2.2.1. Proceso de Compilación

El proceso de compilación recibe una entrada de código fuente en alto nivel generada por parte del usuario y así generar una salida que será una representación intermedia en formato de código de tres direcciones, este formato utilizará sentencias del lenguaje Go para su posterior ejecución.

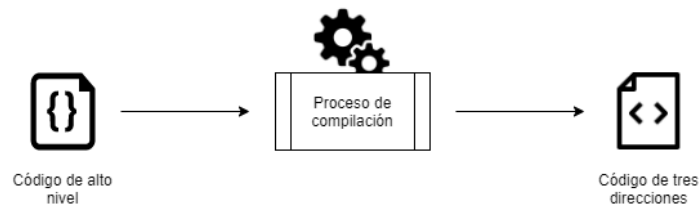


Ilustración 1. Proceso de Compilación.

2.2.2. Proceso de Comprobación

Como el código de tres direcciones utiliza sentencias del lenguaje Go, se debe asegurar que el compilador genere el formato correcto antes de su ejecución, por lo cual, se debe de realizar el proceso de comprobación después de generar el código de tres direcciones. Así para cumplir con los objetivos del proyecto, por este motivo estará disponible un analizador de código de tres direcciones desarrollado por los tutores para constatar que el código generado y optimizado tenga la sintaxis correcta.

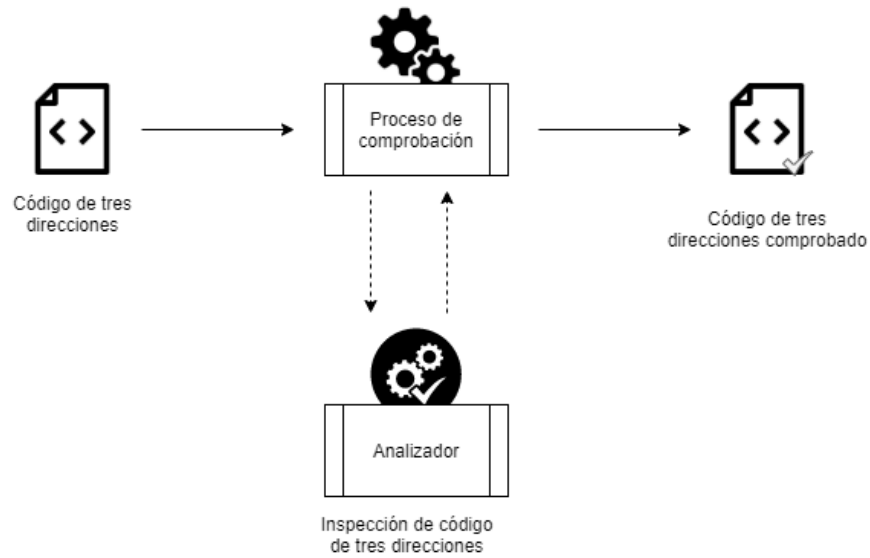


Ilustración 2. Proceso de Comprobación.

2.2.3. Proceso de Ejecución

En el proceso de ejecución recibe como entrada el código de tres direcciones y será ejecutado en un compilador en línea del lenguaje de programación Go, este proceso se puede realizar después del proceso de compilación, se ejecutará el código únicamente si el código de tres direcciones tiene el formato correcto.

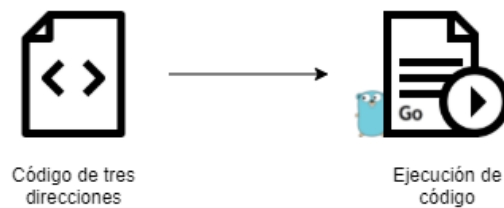


Ilustración 3. Proceso de Ejecución.

2.2.4. Visualización de Reportes

Luego de haber finalizado el proceso de compilación el usuario podrá consultar el reporte de errores y tabla de símbolos.

3. Componentes de la aplicación

A continuación, se describen los componentes de la aplicación.

3.1. Página de Bienvenida

La aplicación debe de contar con una página principal, en donde servirá para identificar al estudiante con sus datos personales, por lo cual, se deberá mostrar los datos tales como el número de carnet, nombre completo y sección a la que pertenece.



Ilustración 4: Página de bienvenida

La aplicación debe tener una barra de menú que permite la navegación entre las páginas y el correcto funcionamiento de la misma, mediante las siguientes opciones:

- **Home:** Redirecciona a la página principal.
- **Editor:** Redirecciona a la página de editor.
- **Compilador:** Se deberá tener opciones para el correcto funcionamiento del compilador, estas opciones se detallan en la página de editor.
- **Reportes:** Se deberá tener opciones para mostrar los distintos reportes generados por el compilador, estas opciones se detallan en la página de reportes.



Ilustración 5: Barra de menú

3.2. Página de Editor

PyTypeCraft tendrá una página que contiene un editor de texto que recibirá como entrada el código fuente de alto nivel y así llevar a cabo el proceso de compilación mostrando como resultado el código de tres direcciones en la consola de salida. Y para los procesos de optimización la entrada será el código de tres direcciones mostrando como resultado el código de tres direcciones optimizado en la consola de salida.

Para este editor no hace falta abrir archivos, basta con copiar y pegar la entrada.

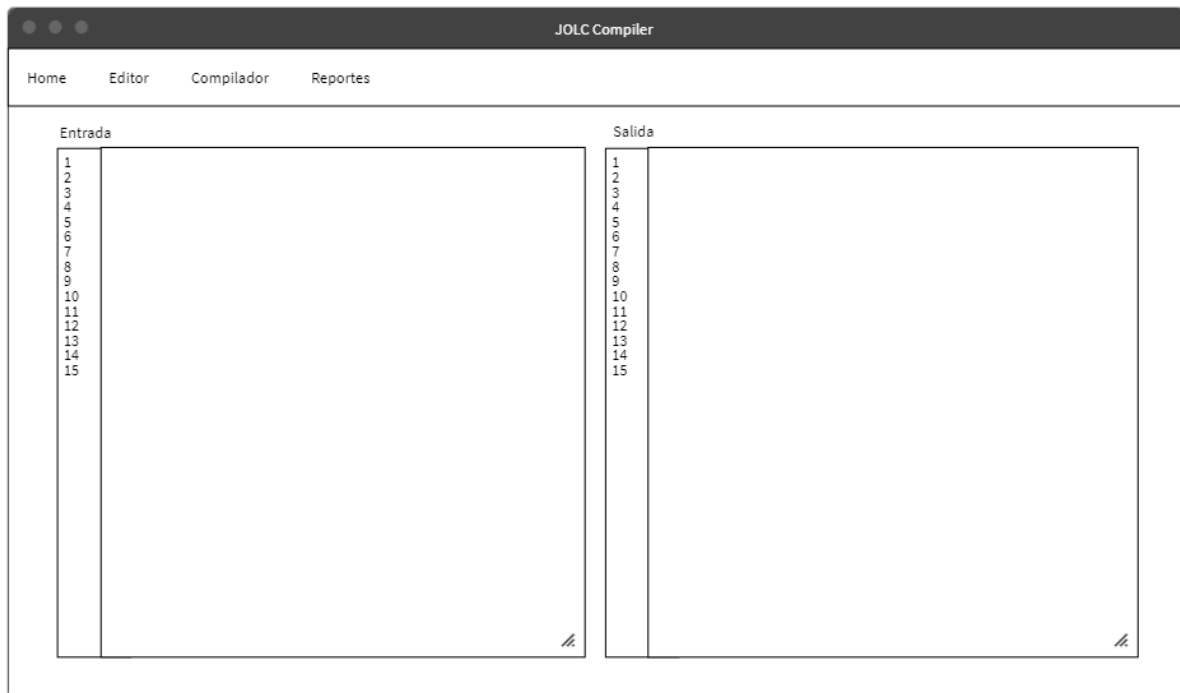


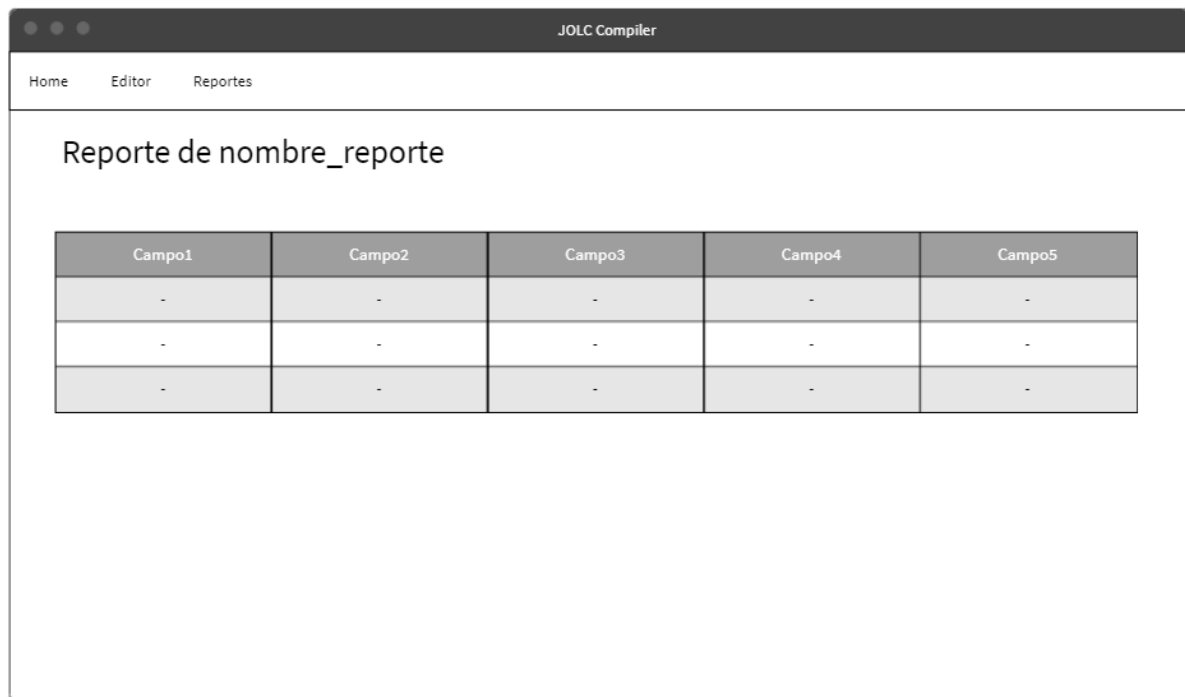
Ilustración 6: Vista de editor

El editor contará con dos opciones que apoyen a la aplicación para realizar los procesos de compilación y optimización, estas opciones son:

- **Compilar:** Realizará el análisis del código fuente para generar el código de tres direcciones.
- **Ejecutar:** Realizará el análisis del código fuente para interpretarlo.

3.3. Página de Reportes

En esta vista se podrán consultar los reportes de tabla de símbolos y tabla de errores después de la compilación, además se puede consultar el reporte de optimización después de haber optimizado el código de tres direcciones.

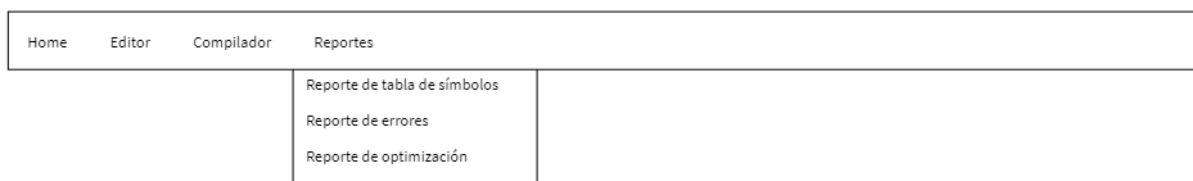


Campo1	Campo2	Campo3	Campo4	Campo5
-	-	-	-	-
-	-	-	-	-
-	-	-	-	-

Ilustración 7: Página de consulta de reportes

Para mostrar un reporte en específico, la aplicación contará con tres opciones, se usará la vista de la ilustración 8 como plantilla para mostrar cada reporte:

- **Reporte de tabla de símbolos:** Este reporte mostrará la tabla de símbolos del código fuente generado en el proceso de compilación.
- **Reporte de errores:** Este reporte mostrará los errores encontrados durante el proceso de compilación.



Home	Editor	Compilador	Reportes
			Reporte de tabla de símbolos Reporte de errores Reporte de optimización

Ilustración 8: Opciones de reportes.

4. Sintaxis de PyTypeCraft

PyTypeCraft provee distintas funcionalidades de Julia, aun así, al tener funciones limitadas de este lenguaje se debe de definir bien la sintaxis de este al ser Julia un lenguaje bastante amplio. También, para conocer más sobre la sintaxis de PyTypeCraft , puede revisar en la documentación de lenguaje de programación TypeScript, pero con las limitaciones que en el proyecto se describe.

4.1. Generalidades

- **Comentarios.** Un comentario es un componente léxico del lenguaje que no es tomado en cuenta en el analizador sintáctico. Pueden ser de una línea (`//`) o de múltiples líneas (`/* ... */`).
- **Case Sensitive.** Esto quiere decir que distinguirá entre mayúsculas y minúsculas.
- **Identificadores.** Un identificador de PyTypeCraft debe comenzar por una letra [A-Za-z] o guión bajo [`_`] seguido de una secuencia de letras, dígitos o guión bajo.
- **Fin de instrucción.** Se hace uso del símbolo `“.”` para establecer el fin de una instrucción.

4.2. Sistema de tipos

PyTypeCraft únicamente aceptará los siguientes tipos de datos, cualquier otro no se deberá tomar en cuenta:

5. **Null:** se representa con la palabra reservada `null`. Indica que no existe ningún valor.
6. **Number:** valores numéricos enteros y decimales. Por ejemplo: 3,2,-1, 3.1416, 1.68...
7. **Boolean:** valores booleanos, true o false.
8. **String:** Cadenas de texto definidas con comillas dobles.
9. **Any:** Cualquier tipo de dato.
10. **Arreglos:** Conjunto de valores indexados entre 0 hasta n. Puede almacenar diferentes tipos. Para más información, consulte la sección 4.9.

```
[10, 20, 30, 40];  
[“Hola”, “Mundo”];  
[‘a’, 2.0, 5, [“Hola”, “Mundo”]];
```

11. **Interface:** Estos son tipos compuestos definidos por el programador. Para mayor detalle, consulte la sección 4.10.

```
interface Rectangulo {  
    base: number;  
    altura: number;  
};
```

11.1.1. Tipos básicos

Son tipos que se definen en el compilador y podemos denominarlos como predefinidos en el lenguaje.

Nombre	Descripción	Expresión de tipo
Nulo	Representación de ausencia de valor.	Nothing
Entero	Representación numérica de enteros. Por ejemplo: 3, 2, 1.	Int64
Decimal	Representación numérica de punto flotante. Por ejemplo: 3.2, 45.6.	Float64
Booleano	Representación lógica. Por ejemplo: true o false.	Bool
Carácter	Representación de carácter, se define con comillas simples. Por ejemplo: 'a'.	Char
Cadena	Representación de cadena de texto definida con comillas dobles.	String

11.1.2. Tipos compuestos

Los constructores de tipo utilizan los tipos básicos para crear nuevos tipos de datos. Los constructores de tipo que utilizan la mayoría de los lenguajes de programación son arreglos y estructuras.

Nombre	Descripción	Expresión de tipo
Arreglos	Conjunto de valores indexados entre 1 hasta n. Puede almacenar diferentes tipos. Para más información, consulte la sección 4.9.	Array
Struct	Estos son tipos compuestos definidos por el programador.	Struct

11.1.3. Comprobación dinámica

Existen ciertos casos en los que no es posible comprobar la validez de operaciones en tiempo de compilación, solamente en tiempo de ejecución. En el proyecto se tomarán en cuenta los siguientes casos:

División entre cero

Se deberá de realizar la comprobación de división entre cero siempre y cuando se realicen expresiones aritméticas con el operador de división (/) o el operador de módulo (%). Se deberá realizar la verificación en código de tres direcciones mostrando como mensaje "MathError". Por ejemplo:

Entrada	Salida
---------	--------

a=(55+3)/(3-3);	<pre> T1 = 55 + 3; T2 = 3 - 3; if (T2 != 0) {goto L1}; fmt.Printf("%c", 77); //M fmt.Printf("%c", 97); //a fmt.Printf("%c", 116); //t fmt.Printf("%c", 104); //h fmt.Printf("%c", 69); //E fmt.Printf("%c", 114); //r fmt.Printf("%c", 114); //r fmt.Printf("%c", 111); //o fmt.Printf("%c", 114); //r T3 = 0; // resultado incorrecto goto L2; L1: T3 = T1 / T2; // resultado correcto L2: </pre>

Índice fuera de los límites

Se deberá realizar la comprobación de índice fuera de los límites, tanto superior como inferior, siempre que se realice un acceso a un arreglo. Se deberá realizar la verificación en código de tres direcciones mostrando como mensaje "BoundsError". Por ejemplo:

Entrada	Salida
<pre> numeros = [1,2,3]; numeros[10]=44; </pre>	<pre> T2 = 10; // índice al que desea acceder if (T2 < 1) {goto L1}; // 1 es el límite inferior del arreglo if (T2 > 3) {goto L1}; // 3 es el límite superior del arreglo goto L2; L1: fmt.Printf("%c", 66) //B fmt.Printf("%c", 111) //o fmt.Printf("%c", 117) //u fmt.Printf("%c", 110) //n fmt.Printf("%c", 100) //d fmt.Printf("%c", 115) //s fmt.Printf("%c", 69) //E fmt.Printf("%c", 114) //r fmt.Printf("%c", 114) //r fmt.Printf("%c", 111) //o fmt.Printf("%c", 114) //r // No continúa con la instrucción goto L3; L2: // Continúa con la instrucción L3: </pre>

Consideraciones:

- En caso se produzca un error al intentar ejecutar una instrucción, esta se debe omitir y continuar con la siguiente instrucción. En caso se produzca un error en una expresión, esta debe resultar con valor 0. Todo esto luego de imprimir en consola el texto solicitado según el caso.

11.2. Expresiones

11.2.1. Aritméticas

Una operación aritmética está compuesta por un conjunto de reglas que permiten obtener resultados con base en expresiones que poseen datos específicos durante la ejecución. A continuación se definen las operaciones aritméticas soportadas por el lenguaje.

Suma

La operación suma se produce mediante la suma de número o strings concatenados.

Operandos	Tipo resultante	Ejemplos
Number + Number	Number	$2 + 3.3 = 5.3$ $2.3 + 8 = 10.3$ $1.2 + 5.4 = 6.6$
String + String <i>Nota: Number puede ser convertido a string con la función nativa "toString" para ser utilizados en esta operación.</i>	String	$"hola" + "mundo" = "holamundo"$ $"Hola" + String(8) = "Hola8"$
string.toUpperCase()	String	$animal = "Tigre";$ $console.log(animal.toUpperCase()); \#TIGRE$
string.toLowerCase()	String	$animal = "Tigre";$ $console.log(animal.toLowerCase()); \#tigre$

Resta

La resta se produce cuando se sustraen el resultado de los operadores, produciendo su diferencia.

Operandos	Tipo resultante	Ejemplos
Number - Number	Number	$2 + 3.3 = -1.3$ $2.3 + 8 = -5.7$ $1.2 + 5.4 = -4.2$

Multiplicación

El operador multiplicación produce el producto de la multiplicación de los operandos.

Operandos	Tipo resultante	Ejemplos
Number * Number	Number	$2 * 3.3 = 6.6$ $2.3 * 8 = 18.4$ $1.2 * 5.4 = 6.48$

División

El operador división se produce el cociente de la operación donde el operando izquierdo es el dividendo y el operando derecho es el divisor.

Operandos	Tipo resultante	Ejemplos
Number / Number	Number	$2 / 3.3 = 0.60$ $2.3 / 8 = 0.2875$ $1.2 / 5.4 = 0.222$

Potencia

El operador de potenciación devuelve el resultado de elevar el primer operando al segundo operando de potencia.

Operandos	Tipo resultante	Ejemplos
Number ^ Number	Number	$2 ^ 3.5 = 11.31$ $2.3 ^ 8 = 783.10$ $1.2 ^ 5.4 = 2.67$

Módulo

El operador módulo devuelve el resto que queda cuando un operando se divide por un segundo operando.

Operandos	Tipo resultante	Ejemplos
Number % Number	Number	$2 \% 3.5 = 2.0$ $2.3 \% 8 = 2.3$ $1.0 \% 5.0 = 1.0$

Nativas

Typescript cuenta con una gran variedad de funciones nativas, sin embargo, PyTypeCraft únicamente contará con las siguientes funciones nativas:

Nombre	Símbolo o función	Descripción	Ejemplo
Aproximación	toFixed(number)	La función formatea un número usando notación de punto fijo.	let n: number = 10.156 n.toFixed(2) // 10.16

Pasar a Exponencial	toExponential	Devuelve una cadena que representa el número en notación exponencial.	let n : number = 123.456 ; n.toExponential(2) ; // "1.23+2"
Convertir a String	toString	Devuelve una cadena que representa cualquier tipo convertido a string	let n : number = 123.456 ; n.toString(); // "123.456"
Convertir a minúsculas	toLowerCase	Convierte todas las letras en minúsculas	let n : string = "HOLA"; n.toLowerCase(); // "hola"
Convertir a mayúsculas	toUpperCase	Convierte todas las letras en mayúsculas	let n : string = "hola"; n.toUpperCase(); // "HOLA"
Separador	split	Divide una cadena en un array de cadenas	let n: string = "hola, mundo"; n.split(","); // ["hola", "mundo"]
Concatenación	concat	Combina 2 o más arrays	let arr: number [] = [1,2,3]; arr.concat([4,5,6]) // [1,2,3,4,5,6]

Agregar	.push	Agrega un dato a un array	<pre>let arr: number [] = []; arr.push(2); arr.push(-4); console.log(arr); // [2,-4]</pre>
Tipo de dato	typeof()	Devuelve el tipo de dato	<pre>let a: number = 1; console.log(typeof(a)); // number</pre>
Longitud	length	Devuelve el tamaño de un array o un string	<pre>let a : string = "Hola"; let b :number = [1,2,3]; console.log(a.length()); // 4 console.log(b.length()); //3</pre>

Operador	Descripción
>	Mayor que: Compara ambos lados y verifica si el izquierdo es mayor que el derecho
<	Menor que: Compara ambos lados y verifica si el derecho es mayor que el izquierdo
>=	Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho
<=	Menor o igual que: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo
===	Igualación: Compara ambos valores y verifica si son iguales
!==	Distinto: Compara ambos lados y verifica si son distintos

EJEMPLOS:

Operandos	Tipo resultante	Ejemplos
Number[>, <, >=, <=] Number String [>, <, >=, <=] String	Bool	4 < 4.3 = true 4.3 > 4 = true 4.3 <= 4.3 = true 4 >= 4 = true "hola" > "hola" = false

1.1.1. Lógicas

Los siguientes operadores booleanos son soportados en PyTypeCraft. No se aceptan valores missing values ni operadores bitwise.

Operación lógica	Operador
OR	
AND	&&
NOT	!

A	B	A && B	A B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

11.3. Impresión

Para mostrar información en la consola o en los reportes, PyTypeCraft cuenta con una forma para imprimir dependiendo de lo que deseamos realizar.

```
console.log("+", "-");
```

Imprime + -

```
console.log("El resultado de 2 + 2 es $(2 + 2)"); # Imprime el resultado de 2 + 2 es 4
console.log("$a $(b[1])"); # Imprime el valor de a y el valor de b[1]
```

1.1. Asignaciones y Declaraciones

Una variable, en PyTypeCraft, es un nombre asociado a un valor. Una variable no puede cambiar su tipo.

La asignación se puede realizar de la siguiente forma:

```
let ID : TIPO = Expresión ;
ó
let ID = Expresión;
```

El sufijo **:TIPO** es opcional. Su función es asegurar que la expresión sea del tipo deseado. En caso la expresión sea distinta al tipo debe marcar un error.

```
let x : number = (3*5); # Correcto
let str : number = "Saludo"; # ERROR: expected Number, got String
let var1 : string = true; # ERROR: expected String, got Bool
let var = 1234; # Correcto (aquí se asigna implícitamente el tipo number)
```

Para el manejo de variables “locales” y “globales” se deberá de distinguir el límite del bloque en el que se encuentra declarada y así determinar si la variable es global o solamente local

```

1 // Ejemplo 1: Entornos. Variables globales y locales.
2 let x:number = (3*5);// 15
3 let str = "Saludo";
4
5 function ejemplo(){
6     str="Ejemplo";// PyTypeCraft hace referencia a la variable global str
7     let x = 0; // PyTypeCraft crea una nueva variable local
8     for (let i = 0; i < 10; i++){
9         let x: number;// Creando variable local.
10        x = i * 2; // Gracias a la variable 'local' no hace referencia a la variable de la línea 6
11        console.log(x);// Imprime: 2 4 6 8 10...
12    }
13    console.log(x);// 0 --> la variable nunca fue modificada
14 }
15
16 ejemplo();
17 console.log(x);// 15
18 console.log(str);// Ejemplo --> Modificada dentro de ejemplo()

```

```

1 // Ejemplo 2: Aclaraciones de alcance de variables
2 let x = 15;
3 let y = 44;
4
5 function ejemplo2() {
6     y = 5; // Se modifica la variable global
7     console.log(x); // Primero busca en el entorno local y luego en el global
8 };
9
10 ejemplo2();
11
12 console.log(x);
13 console.log(y);

```

```

1 let x = 3;
2
3 function ejemplo3() {
4     for (let i = 0; i < 5; i++) {
5         let x = null; // Aquí 'x' es local para el bloque del bucle for y está inicializado con 'null'.
6         console.log(x); // Esto imprimirá 'null' en cada iteración del bucle.
7     }
8 }
9
10 ejemplo3();

```

1.2. Funciones

1.2.1. Creación de funciones

Las funciones en PyTypeCraft se crean con la palabra clave *function* seguida del nombre de la función y, entre paréntesis, los parámetros de entrada de la función. En PyTypeCraft es obligatorio utilizar la instrucción *return* para retornar un valor. En caso no se utilice o se utilice *return* sin valor, la función devolverá nada (el dato null).

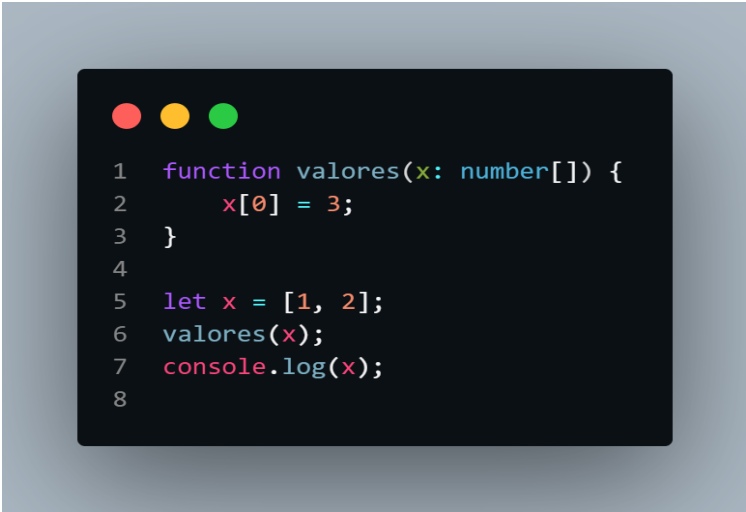
```
function NOMBRE_FUNCION (LISTA_PARAMETROS) {  
  LISTA_INSTRUCCIONES  
};
```

1.2.2. Llamada a funciones

La llamada a funciones se realiza con el nombre de la función, y entre paréntesis, los parámetros a pasar.

1.2.3. Paso por valor o por referencia

En PyTypeCraft, los únicos tipos que son pasados por referencia son los arreglos y struct, por lo que si se modifican dentro de una función también se modificarán fuera. El resto de tipos son pasados por valor.



```
1  function valores(x: number[]) {  
2    x[0] = 3;  
3  }  
4  
5  let x = [1, 2];  
6  valores(x);  
7  console.log(x);  
8
```

1.3. Condicionales

El lenguaje PyTypeCraft cuenta con sentencias condicionales, la evaluación condicional permite que porciones de código se evalúen o no se evalúan dependiendo del valor de una expresión booleana. Estos se definen por las instrucciones *if*, *elseif*, *else*.

Consideraciones:

- Las instrucciones *else if* y *else* son opcionales.
- La instrucción *else if* se puede utilizar tantas veces como se desee.

```

1  let x = 8; // Definimos un valor para 'x', se necesitará para ejecutar el código
2
3  // Instrucción if
4  if (x === 8) {
5      let var1 = x + 8;
6      console.log(var1.toString());
7  }
8
9  // Instrucción if, elseif, else
10 if (x === 8) {
11     let var1 = x + 8;
12     console.log(var1.toString());
13 } else if (x < 8) {
14     let var1 = x / 3;
15     console.log(var1.toString());
16 } else {
17     console.log("Error");
18 }
19
20 // Instrucción if, else
21 if (x === 10) {
22     let var2 = x + 10;
23     console.log(var2.toString());
24 } else {
25     console.log(x+8);
26 }

```

1.4. Loops

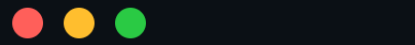
En el lenguaje PyTypeCraft existen dos sentencias iterativas, este tipo de sentencias son aquellas que incluyen un bucle sobre una condición, las sentencias iterativas que soporta el lenguaje son las siguientes:

1.4.1. Ciclo while

Esta sentencia ejecutará todo el bloque de sentencias solamente si la condición es verdadera, de lo contrario las instrucciones dentro del bloque no se ejecutarán, seguirá su flujo secuencial.

Consideraciones:

- Si la condición es falsa, detendrá la ejecución de las sentencias de la lista de instrucciones.
- Si la condición es verdadera, ejecuta todas las sentencias de su lista de instrucciones.



```

1  let var1 = 0;
2  while (var1 < 10) {
3      console.log(var1);
4      var1 = var1 + 1;
5  }

```

1.4.2. Ciclo for

Esta sentencia puede iterar sobre un rango de expresiones, cadena de caracteres (*"string"*) o arreglos. Permite iniciar con una variable como variable de control en donde se verifica la condición en cada iteración, luego se deberá actualizar la variable en cada iteración.

Consideraciones:

- Contiene una variable declarativa que se establece como una variable de control, esta variable servirá para contener el valor de la iteración.
- La expresión que evaluará en cada iteración es de tipo rango, string o array. Aunque también se puede especificar mediante una variable.



```

1  // Recorre rango de 1:4
2  for (let i = 1; i <= 4; i++) {
3      console.log(i, " ");
4  }
5
6  // Recorre las letras de la cadena
7  for (let letra of "Hola Mundo!") {
8      console.log(letra, "-");
9  }
10
11 let cadena = "OLC2";
12 for (let letra of cadena) {
13     console.log(letra, "-");
14 }
15
16 for (let animal of ["perro", "gato", "tortuga"]) {
17     console.log(`${animal} es mi favorito`);
18 }
19
20 let arr = [1,2,3,4,5];
21 for (let i = 1; i < 4; i++) { // los índices en JavaScript y TypeScript comienzan en 0, no en 1
22     c

```

1.4.3. Sentencias de transferencia

A veces es conveniente terminar un ciclo antes de que la condición sea falsa o detener la iteración de una sentencia loop antes de que se alcance el final del objeto iterable, además también es conveniente saltar unas sentencias de un ciclo en determinadas ocasiones y por para las funciones es necesario el retorno de un valor.

- Break
- Continue
- Return

Consideraciones:

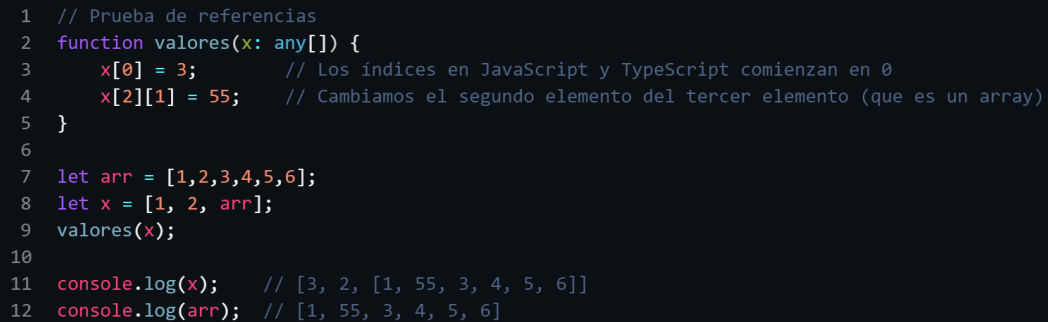
- Se debe validar que la sentencia *break* y *continue* se encuentre únicamente dentro de una sentencia loop.
- Es necesario validar que la sentencia *break* detenga las sentencias asociadas a una sentencia loop.
- Es necesario validar que la sentencia *continue* salte a la siguiente iteración asociada a su sentencia loop.
- Es requerido validar que la sentencia *return* esté contenida únicamente en una función.

```
1 // Ejemplo break
2 while (true) {
3     console.log(true); // Imprime true solo una vez
4     break;
5 }
6
7 // Ejemplo continue
8 let num = 0;
9 while (num < 10) {
10     num = num + 1;
11     if (num === 5) {
12         continue;
13     }
14     console.log(num); // Imprime 1 2 3 4 6 7 8 9 10
15 }
16
17 // Ejemplo de return
18 function funcion() {
19     let num = 0;
20     while (num < 10) {
21         num = num + 1;
22         if (num === 5) {
23             return 5;
24         }
25         console.log(num);
26     }
27     return 0;
28 }
29
30 console.log(funcion()); // Para ver el valor que retorna la función
31
```


1.5. Arreglos

En PyTypeCraft, se cuenta este tipo de dato compuesto y mutable. Puede contener cualquier tipo de dato.

Además, toma en cuenta que al tratarse de un tipo mutable, maneja referencias, por ejemplo:



```
1 // Prueba de referencias
2 function valores(x: any[]) {
3     x[0] = 3;           // Los índices en JavaScript y TypeScript comienzan en 0
4     x[2][1] = 55;       // Cambiamos el segundo elemento del tercer elemento (que es un array)
5 }
6
7 let arr = [1,2,3,4,5,6];
8 let x = [1, 2, arr];
9 valores(x);
10
11 console.log(x);        // [3, 2, [1, 55, 3, 4, 5, 6]]
12 console.log(arr);      // [1, 55, 3, 4, 5, 6]
```

1.6. Structs

Los *structs* son tipos compuestos que se denominan registros, los tipos compuestos se introducen con la palabra clave *interface* seguida de un bloque de nombres de campos, opcionalmente con tipos usando el operador “.”.

Consideraciones:

- Los atributos sin especificar el tipo, en consecuencia pueden contener cualquier tipo de valor.
- Los *structs* también se pueden utilizar como retorno de una función.
- Las declaraciones de los *structs* se pueden utilizar como expresiones.
- Los atributos se pueden acceder por medio de la notación “.”.



```
1  interface Carro {
2      placa: string;
3      color: string;
4      tipo: string;
5  }
6
7  let c1: Carro = {
8      placa: "090PLO",
9      color: "gris",
10     tipo: "mecanico"
11 };
12
13 let c2: Carro = {
14     placa: "P0S921",
15     color: "verde",
16     tipo: "automatico"
17 };
18
19 // Asignación Atributos
20 c1.color = "cafe";    // Cambio aceptado
21 c2.color = "rojo";    // Cambio aceptado
22
23 // Acceso Atributo
24 console.log(c1.color); // Imprime cafe
```

12. Generación de Código Intermedio

El código intermedio es una representación intermedia del programa fuente que se ingresó en PyTypeCraft. Esta representación intermedia se realizará en código en tres direcciones, las cuales son secuencias de pasos de programa elementales.

En el proyecto se utilizará las sentencias del lenguaje Go para escribir el código tres direcciones, manejando este lenguaje con limitaciones para que se apliquen correctamente los conceptos de generación de código tres direcciones aprendido en la clase magistral.

El código tres direcciones que genera PyTypeCraft se detalla en esta sección.

No está permitido el uso de toda función o característica del lenguaje Go no descrita en este apartado. Se utilizará una herramienta de análisis para verificar que el código de tres direcciones generado tenga el formato correcto.

12.1. Tipos de dato

El lenguaje a compilar solo acepta tipos de datos numéricos, es decir, tipos int y float.

Consideraciones:

- No está permitido el uso de otros tipos de datos como cadenas o booleanos.
- El uso de arreglos no está permitido, únicamente para las estructuras heap y stack que se explican con más detalle más adelante.
- Por facilidad, se recomienda trabajar todas las variables de tipo float.

12.2. Temporales

Los temporales serán creados por el compilador en el proceso de generación de código de tres direcciones. Estas serán variables de tipo float. El identificador asociado a un temporal puede ser de la siguiente manera.

```
t[0-9]+  
Ej.  
t1  
t145
```

12.3. Etiquetas

Las etiquetas son identificadores únicos que indican una posición en el código fuente, estas mismas serán creadas por el compilador en el proceso de generación de código en tres direcciones. El identificador asociado a una etiqueta puede ser de la siguiente manera.

```
L[0-9]+  
Ej.  
L1  
L21
```

Las operaciones aritméticas contarán con:

- Resultado
- Argumento 1
- Argumento 2
- Operador

Operación	Símbolo	Ejemplo
Suma	+	t1=t0+1
Resta	-	t2=50-12
Multiplicación	*	t33=5*5
División	/	t67=4/1
Módulo	%	t44=4 % 2

Comentarios

Para llevar un mejor control de las instrucciones que se realizan dentro del código tres direcciones se recomienda el uso de comentarios donde se podrá definir el flujo de cada bloque de código, los comentarios se definen de la siguiente manera:

- Comentarios de una línea. Inician con un conjunto de barras diagonales (//) y continúan hasta el final de la línea.
- Comentarios de múltiples líneas. Inician con los símbolos “/*” y finalizan con los símbolos “*/”

12.4. Saltos

Para definir el flujo que seguirá el programa se contará con bloques de código, estos bloques están definidos por etiquetas. La instrucción que indica que se realizará un salto hacia una etiqueta es la palabra reservada “goto”.

En el proyecto se utilizarán los dos formatos de saltos que son:

- Condicional. Se realiza una evaluación para determinar si se realiza el salto.
- No condicional. Realiza el salto sin realizar una evaluación.

12.4.1. Saltos no condicionales

El formato de saltos no condicionales contará únicamente con una instrucción **goto** que indicara una etiqueta destino específica, en la cual se continúa con la ejecución del programa.

//Ejemplo de salto no condicional

```
goto L1
fmt.Printf("%c", 64) //código inalcanzable
L1:
t2 = 100 + 5
```

12.4.2. Saltos condicionales

El formato de los saltos condicionales utilizará la instrucción `if` del lenguaje Go donde se realizará un salto a una etiqueta donde se encuentre el código a ejecutar si la condición es verdadera, seguida de otro salto a una etiqueta donde están las instrucciones si la condición no se cumple.

Las instrucciones `if` tendrán como condición una expresión relacional, dichas expresiones se definen en la siguiente tabla:

Operación	Símbolo	Ejemplo
Menor que	<	t3<4
Mayor que	>	t6>44
Menor o igual que	<=	t55<=50
Mayor o igual que	>=	t99>=100
Igual que	==	t23==t44
Diferente que	!=	t34!=99

```
//Ejemplo de saltos condiciones
if (10 == 10) {goto L1}
goto L2
L1:
//código si la condición es verdadera
L2:
//código si la condición es falsa
```

12.5. Asignación a temporales

La asignación nos va a permitir cambiar el valor de los temporales, para lograrlo se utiliza el operador igual, este permite una asignación directa o con una expresión.

```
//Entrado código alto nivel
print(1+2*5);

//Salido código en tres direcciones en lenguaje Go
t1 = 2 * 5
t2 = 1 + 1
```

```
fmt.Printf("%d", int(t2))
```

12.6. Métodos

Estos son bloques de código a los cuales se accede únicamente con una llamada al método.

```
//Definición de métodos
func x() {
    goto L0
    fmt.Printf("%d", int(100))
L0:
    return
}
```

Consideraciones:

- No está permitido el uso de parámetros en los métodos. Debe utilizar el stack para el paso de parámetros.
- Al final de cada método se debe incluir la instrucción "return".

12.7. Llamada a métodos

Esta instrucción nos permite invocar a los métodos. Al finalizar su ejecución se retorna el control al punto donde fue llamada para continuar con las siguientes instrucciones.

```
func funcion1(){
    fmt.Printf("%d", int(100))
    return
}

func main(){
    // INSTRUCCIONES DE MAIN
    funcion1();                // Llamada a método funcion1
    // DESPUÉS DE EJECUTAR funcion1 REGRESA A MAIN
}
```

12.8. Impresión en consola

Su función principal es imprimir en consola un valor, el primer parámetro que recibe la función es el formato del valor a imprimir, y el segundo es el valor en sí.

La siguiente tabla lista los parámetros permitidos para el proyecto:

Parámetro	Acción
%c	Imprime el carácter del identificador, se basa según el código ASCII.
%d	Imprime valores enteros. El segundo parámetro debe ser una conversión explícita de int.
%f	Imprime valores con punto decimal.

```
fmt.Printf("%d", int(100)) // Imprime 100
fmt.Printf("%c", 36)      // Imprime $
fmt.Printf("%f", 32.2)    // Imprime 32.200000
```

12.9. Estructuras en tiempo de ejecución

El proceso de compilación genera el código en tres direcciones, el cual se ejecutará en un compilador de GO separado. En el código de tres direcciones no existen cadenas, operaciones complejas, llamadas a métodos con parámetros y otras características que sí están presentes en los lenguajes de alto nivel.

En el proyecto se utilizarán las siguientes dos estructuras:

- Stack (pila)
- Heap (montículo)

Estas estructuras se utilizarán para almacenar los valores que sean necesarios durante la ejecución.

12.9.1. Stack

También conocido como pila de ejecución, es una estructura que se utiliza para guardar los valores de las variables locales, así como también los parámetros y el valor de retorno de las funciones en alto nivel.

Esta estructura utilizará un apuntador llamado "Stack Pointer", que se identifica con el nombre **P**, este valor va cambiando conforme se ejecute el programa, y su manejo debe ser cuidadoso para no corromper espacios de memoria ajenos al método que se está ejecutando, su asignación se realizará de la misma manera que se realizan las asignaciones temporales.

```
var stack [100000]float64 // Stack
var P float64             // Stack Pointer

P = P + 5; // Cambio de ámbito
P = P - 5; // Regreso a ámbito

stack[int(P)] = 10
t1 = stack[int(P)]
```

12.9.2. Heap

También conocido como montículo, es una estructura de control del entorno de ejecución encargada de guardar las referencias a las cadenas, arreglos y estructuras. Esta estructura también cuenta con un apuntador que se identifica con el nombre **H**.

A diferencia del apuntador **P**, este apuntador no decrece, sino que sigue aumentando su valor, su función es apuntar a la primera posición de memoria libre dentro del heap.

```
var heap [100000]float64 // Heap
var H float64           // Heap Pointer

H = H + 1;
T1 = H

heap[int(P)] = 10
t1 = heap[int(P)]
```

Consideraciones:

- Al guardar cadenas, cada espacio debe ser ocupado por únicamente un carácter representado por su código ASCII.
- El heap solamente crece, nunca reutiliza espacios de memoria.

12.9.3. Acceso y asignación a estructuras en tiempo de ejecución

Para realizar las asignaciones y el acceso a estas estructuras, se debe respetar el formato de código de 3 direcciones:

- La asignación a las estructuras se debe realizar por medio de un temporal o un valor puntual, no es permitido el uso de operaciones aritméticas o lógicas para la asignación a estas estructuras.
- No se permite la asignación a una estructura mediante el acceso a otra, por ejemplo "Stack[0] = Heap[100]".

```
//Asignación
Heap[int(H)] = t1
<código>
Stack[int(t2)] = 150
//Acceso
t10 = Heap[int(t10)]
t20 = Stack[int(t150)]
```

12.10. Encabezado

En esta sección se definirán todas las variables y estructuras a utilizar para que al momento de ejecutar el código de tres direcciones funcione correctamente, puesto que, es necesario

el uso de la librería `fmt` para la instrucción de imprimir y la declaración de variables antes de usar alguna variable en el código. Únicamente en esta sección se permite el uso de declaraciones en Go, no es permitido realizar declaraciones dentro de métodos. El encabezado debe ser generado junto con el código tres direcciones para hacer uso de los temporales y las estructuras necesarias.

La estructura del encabezado es la siguiente:

```
package main
import ( "fmt" )           // importar para el uso de printf

var stack [1000]float64    // estructura Stack
var heap [1000]float64    // estructura Heap
var P, H float64          // declaración de Stack y heap pointer
var t1, t2, t3 float64    // declaración de temporales
```

Consideraciones:

- No es permitido el uso de otras librerías ajenas a “`fmt`”
- Todas las declaraciones de temporales se deben encontrar en el encabezado
- El tamaño que se le asigne al stack y heap queda a discreción del estudiante. Tomar en cuenta que el heap únicamente aumenta, por lo que el tamaño de este debe ser grande.

12.11. Método main

Este es el método donde iniciará la ejecución del código traducido. Su estructura es la siguiente:

```
func main(){
    // instrucciones
}
```

12.12. Comprobación de código tres direcciones

Una vez generado el código tres direcciones este será ingresado a un analizador de la sintaxis para corroborar que el código generado sea correcto y no se encuentre código diferente al explicado anteriormente, una vez analizado se procederá a ejecutar el código en tres direcciones en un compilador de GO para obtener el resultado esperado.

13. Reportes generales

13.1. Reporte de Tabla de Símbolos

En este reporte se solicita mostrar la tabla de símbolos después de la compilación de la entrada. Se deberán mostrar todas las variables, funciones y struct reconocidas, junto con su tipo y toda la información que el estudiante considere necesaria. Este reporte al menos debe contener la fila y columna de la declaración del símbolo junto con su nombre, tipo y ámbito. En el caso de las funciones, deberá mostrar el nombre de sus parámetros, en caso tenga.

Nombre	Tipo	Ámbito	Fila	Columna
x		valores	2	18
valores	Función	Global	2	1
arr	arreglo	Global	6	1
x	arreglo	Global	7	1

13.2. Reporte de Tabla de errores

Su aplicación deberá ser capaz de detectar y reportar todos los errores semánticos que se encuentren durante la compilación. Su reporte debe contener como mínimo la siguiente información.

- Descripción del error.
- Número de línea donde se encontró el error.
- Número de columna donde se encontró el error.
- Fecha y hora en el momento que se produce un error.

No.	Descripción	Línea	Columna	Fecha y hora
1	El struct Persona no fue declarado	112	15	14/8/2021 20:16
2	El tipo string no puede multiplicarse con un real	80	10	14/8/2021 20:16
3	No se esperaba que la instrucción break estuviera fuera de un ciclo.	1000	5	14/8/2021 20:16

14. Manejo de errores

14.1. Errores semánticos

El compilador deberá ser capaz de detectar todos los errores semánticos que se encuentren durante el proceso de compilación. Todos los errores se deberán de recolectar y se mostrará un reporte de errores antes mencionado.

Un error semántico es cuando la sintaxis es la correcta, pero la lógica no es la que se pretendía, por eso, la recuperación de errores semánticos será de ignorar la instrucción en donde se generó el error.

Si se detecta cualquier tipo de error semántico el estudiante deberá descartar la instrucción completa, Un error semántico se dará por ejemplo al intentar usar una variable que no ha sido declarada.

```
...  
int numero1 = numero2 + 10;  
...
```

Como se ve en ejemplo la variable "numero2" no ha sido declarada por lo que se mostrará un error semántico, la forma en que se manejan los errores semánticos consistirá en descartar la instrucción la cual contiene el error y se reportará el error de en el reporte de errores antes mencionado.

15. Entregables y Calificación

Para el desarrollo del proyecto se deberá utilizar un repositorio de GitHub, este repositorio deberá ser privado y tener a los auxiliares como colaboradores.

15.1. Entregables

El código fuente del proyecto se maneja en GitHub por lo tanto, el estudiante es el único responsable de mantener actualizado dicho repositorio hasta la fecha de entrega, si se hacen más commits luego de la fecha y hora indicadas no se tendrá derecho a calificación.

- Código fuente y archivos de compilación publicados en un repositorio de GitHub cada uno en una carpeta independiente.
- Enlace al repositorio y permiso al auxiliar para acceder. Para darle permiso al auxiliar, agregar este usuario al repositorio
 - DiiAns23
- Aplicación web con la funcionalidad del proyecto publicada en AWS.

15.2. Restricciones

- La herramienta para generar los analizadores del proyecto será Python PLY. La documentación se encuentra en el siguiente enlace <https://www.dabeaz.com/ply/>.
- No está permitido compartir código con ningún estudiante. Las copias parciales o totales tendrán una nota de 0 puntos y los responsables serán reportados a la Escuela de Ingeniería en Ciencias y Sistemas.
- El resultado final del proyecto debe ser una aplicación web funcionando en AWS, no será permitido descargar el repositorio y calificar localmente.
- El desarrollo y entrega del proyecto es en parejas.

15.3. Consideraciones

1. Es válido el uso de cualquier Framework para el desarrollo de la aplicación siempre y cuando la aplicación final pueda ser publicada en AWS.
2. El repositorio únicamente debe contener el código fuente empleado para el desarrollo, no deben existir archivos PDF o DOCX.
3. El sistema operativo a utilizar es libre.
4. El lenguaje está basado en Typescript, por lo que el estudiante es libre de realizar archivos de prueba en esta herramienta, el funcionamiento debería ser el mismo y limitado a lo descrito en este enunciado.

4.1. Calificación

- La calificación se realizará dentro de la máquina de los auxiliares, ya que es muy importante que tengan la última versión de su proyecto subida a Heroku y las rutas definidas anteriormente.
- Se probará que el estudiante genere el compilado correcto y que esté siendo ejecutado en AWS.
- Durante la calificación se realizarán preguntas sobre el código y reportes generados para verificar la autoría de este, de no responder correctamente la mayoría de las preguntas se reportará como copia.
- Se tendrá un máximo de 45 minutos por estudiante para calificar el proyecto.
- La hoja de calificación describe cada aspecto a calificar, por lo tanto, si la funcionalidad a calificar falla en la sección indicada se tendrá 0 puntos en esa

funcionalidad y esa nota no podrá cambiar si dicha funcionalidad funciona en otra sección.

- Si una función del programa ya ha sido calificada, esta no puede ser penalizada si en otra sección la función falla o es errónea.
- Los archivos de entrada permitidos en la calificación son únicamente los archivos de pruebas preparados por los tutores.
- Los archivos de entrada podrán ser modificados solamente antes de iniciar la calificación eliminando funcionalidades que el estudiante indique que no desarrolló.
- Los archivos de entrada podrán ser modificados si contienen errores semánticos no descritos en el enunciado o provocados para verificar el manejo y recuperación de errores.

4.2. Entrega de proyecto

- La entrega será mediante GitHub, y se va a tomar como entrega el código fuente publicado en el repositorio a la fecha y hora establecidos.
- Cualquier commit luego de la fecha y hora establecidas invalidará el proyecto, por lo que se calificará hasta el último commit dentro de la fecha válida.
- Fecha de entrega:

ENTREGA: 27 DE JUNIO DE 2023 ANTES DE LAS 08:00