Objetivos Generales

- Familiarizar al estudiante con las herramientas JFlex y CUP.
- Aplicar conocimientos de análisis léxico, sintáctico y semántico.
- Comprender el funcionamiento de un árbol de sintaxis abstracta y la tabla de símbolos.

Objetivos Específicos

- Crear archivos de configuración con JFlex y CUP.
- Combinar la funcionalidad de JFlex y CUP en aplicaciones reales.

Descripción de la Actividad

En la actualidad con la infinidad de lenguajes de programación y con el desarrollo de tecnologías o lenguajes un poco más flexibles como Python o JavaScript dónde la curva de aprendizaje es más sencilla y donde se declara una variable sin indicar el tipo y se puede cambiar el tipo de dato de una variable o no se sabe con exactitud qué tipo de dato retorna una función, pareciera para algunos programadores una buena opción sobre los lenguajes fuertemente tipados cuya curva de aprendizaje es más difícil y son de sintaxis un poco más estricta. Sin embargo para otro buen grupo de programadores los lenguajes fuertemente tipados ofrecen una serie de ventajas como la detección temprana de errores y la mejora en calidad de código.

Además los lenguajes fuertemente tipados como TypeScript han evolucionado para incorporar características que los hacen más flexibles y fáciles de usar, por ejemplo TypeScript permite usar tipos de datos opcionales, ayudando a los programadores a ser más eficientes sin sacrificar la seguridad y la calidad del código.

En resumen los lenguajes de programación fuertemente tipados seguirán siendo una parte importante del panorama de la programación debido a sus ventajas en términos de seguridad, calidad del código y detección temprana de errores.

En base a lo anterior, la empresa “App Design” lo ha contratado para la creación de un nuevo lenguaje de programación fuertemente tipado que se asemeje a TypeScript, llamado TypeSecure (ts) Cómo programador experimentado estará a cargo de desarrollar el lenguaje desde cero, asegurándose de que cumpla con los estándares de calidad y funcionalidad requeridos.

Su trabajo será crear un compilador y desarrollar un conjunto de herramientas de desarrollo que faciliten la programación en este nuevo lenguaje asegurando un buen rendimiento del lenguaje.

Sintaxis Básica de TypeSecure

Tipos de Datos:

La siguiente tabla describe los tipos de datos que estarán disponibles en TypeSecure:

Tipo de dato	Palabra reservada	Descripción
Number	number	Valores de coma flotante de 64 bits de precisión “double”.
BigInt	bigint	Valores enteros de 64 bits
String	string	Representa una secuencia de caracteres unicode, entre comillas dobles o simples.
Boolean	boolean	Representa valores lógicos (true o false)
Void	void	Se utiliza en tipos de devolución de funciones, para todas aquellas que no devuelven ningún valor.
Undefined	undefined	Denota el valor dado a todas las variables no inicializadas.

Declaración de variables:

La sintaxis para declarar variables es usando las palabras reservadas **const** y **let** seguida del nombre de la variable, seguida de dos puntos y el tipo de la variable. Ejemplos de declaración y/o asignación de variables:

```
// declaración de variables
const TOTAL: number = 100;
let name: string = 'Jonh';
let score1: number = 20, lastname: string = 'Perez Leon';
let _flag: boolean = score1 > TOTAL;
let entero1: bigint = 144n;
let _flag2: boolean; //  _flag2 tiene el valor de undefined
const SUMA = 1000;// se asume que SUMA es de tipo number
```

const: Usado para declarar constantes, es decir a esta variable no se le puede asignar un nuevo valor y siempre se le debe de asignar un valor.

let: Usado para declarar variables que se les puede asignar otro valor.

El nombre de las variables debe de empezar con cualquier letra minúscula o mayúscula o guiones bajos seguido de cualquier carácter alfanumérico (letras mayúsculas y/o minúsculas, números y guiones bajos).

Inferencia de tipos:

Dado que TypeSecure es fuertemente tipado, es posible declarar variables sin indicar el tipo y TypeSecure infiere el tipo de la variable según el valor asignado, para estos casos la variable siempre debe de tener un valor asignado, en caso contrario se debe indicar un error semántico.

```
/* inferencia de tipos */
const name = 'Jose Perez Leon'; // name es de tipo string
let __value = name != 'Luis Miguel'; // __value es de tipo boolean
let enterito = 125n + 2n; // enterito es de tipo bigint
let amount = 12, quantity = 100.5; // amount y quantity son de tipo number
```

Cómo se puede notar, para asignar un valor a una variable de tipo ***bigint*** es valor o número a asignar tener el sufijo *n*.

Operaciones aritméticas

Operador	Descripción	Ejemplo
+	Suma, retorna la suma de los operandos	a + 1
-	Resta, retorna la diferencia de los operandos	b - 3
/	División, ejecuta la división y devuelve el cociente	a / 100
%	Módulo, ejecuta la división y devuelve el resto.	a % b
++	Incrementa el valor de la variable en 1	a++

--	Decrementa el valor de la variable en 1	b--
()	Paréntesis	a * (b + 1)

Los operadores incremento (++) y decremento (--) són operadores unarios, es decir que solo necesitan una variable para ejecutarse y no se pueden efectuar sobre valores constantes (y variables de tipo const).

El resto de operadores respeta la precedencia que se sigue en cualquier lenguaje de programación y en específico en TypeScript.

Importante:

- Los operadores aritméticos sólo pueden aplicarse a los tipos *number* y *bigint*, y no es posible efectuar una operación aritmética entre dos variables de tipos distintos. Es decir, no es posible efectuar la operación 101 + 1n, porque 101 es de tipo *number* y 1n es de tipo *bigint*.

- Para efectuar una operación entre dos variables de distinto tipo será necesario convertir ambas variables a tipo *bigint* o ambas variables a tipo *number*, para ello se usarán las funciones *Number()*, *BigInt()*, *String()* y *Boolean()* que se explican más adelante.

Operadores Relacionales

Los operadores relacionales prueban o definen el tipo de relación entre dos entidades. Los operadores relacionales devuelven un valor booleano, es decir: ***true*** o ***false***.

Operador	Descripción
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual a
!=	Distinto de ó no igual que

Operadores Lógicos

Los operadores lógicos se utilizan para combinar dos o más condiciones. Los operadores lógicos también devuelven un valor booleano. Suponiendo que a es 10 y b es 20:

Operador	Descripción	Ejemplo
&&	El operador AND devuelve verdadero si todas las expresiones evaluadas son verdaderas.	(a > 10 && b > 10) es ***false***
||	El operador OR devuelve verdadero si alguna de las expresiones evaluadas es verdadera.	(a > 10 || b > 10) es ***true***
!	El operador NOT devuelve el inverso de la expresión evaluada	!(a > 10) es ***true***

Operador de concatenación (+):

El operador de concatenación es aplicable cuando se quiere unir dos strings. Pero tambien es aplicable cuando al menos uno de los operadores es de tipo string.

```
// contanación
const greeting = "Hello" + " world";
const message = 'El total es: ' + 100n
const message2 = "El resultado es: " + greeting == 'Hello world';
console.log(greeting); // Hello world
console.log(message); // El total es: 100
console.log(message2); // El resultado es: true
```

Función Number

Valores de otros tipos pueden ser convertidos al tipo *number* usando la función *Number()*, esta función se puede resumir como sigue:

- Datos de tipo *number*, son retornados tal cual.
- Datos de tipo *bigint*, son convertidos a datos de tipo *number*.
- Para las expresiones u operaciones de tipo *boolean*: *true* es convertido a 1; *false* a 0.
- *Strings* se convierten analizandolas y verificando que contengan un número literal

(number literal). Aspectos a tomar en cuenta para esto:
    - Espacios en blanco al inicio y al final son ignorados.
    - Los símbolos + y - están permitidos al comienzo de la cadena para indicar el signo. Sin embargo, el signo sólo puede aparecer una vez y no debe ir seguido de un espacio en blanco.
    - Cadenas vacías y espacios en blanco son convertidos a cero.
- En caso de no poder convertir el valor pasado a la función *Number*, se debe lanzar un error indicando fila y columna de la instrucción y la razón por la cuál no es posible realizar la conversión.

```
console.log(Number(12n) == 12) // output is true

let a = 100n, b = 10;
let div = Number(a) / b;
console.log(div); // output is 10

const flag = div == 10;// true
if (Number(flag) == 1) {// true
    console.log('it works');
}

if (Number(" -12.23 ") == -12.23) {// true
    console.log('it works again');
}
```

Función BigInt

Valores de otros tipos pueden ser convertidos al tipo *bigint* usando la función *BigInt()*, que puede resumirse como sigue:

- Valores u operaciones de tipo *bigint*, son retornados tal cuál.
- Valores u operaciones booleanas con valor *true* son convertidas a 1n y con valor *false* a 0n.
- Valores de tipo *number* son convertidos a *bigint*, y cuando estos tengan un valor decimal se debe de lanzar un error indicando que no es posible la conversión por “pérdida de precisión”, además de indicar la fila y columna de la instrucción.
- *Strings* se convierten verificando que contengan un *bigint literal*, es decir:
    - Los espacios al inicio y al final son ignorados.
    - Espacios en blanco al inicio y al final son ignorados.
    - Los símbolos + y - están permitidos al comienzo de la cadena para indicar el signo. Sin embargo, el signo sólo puede aparecer una vez y no debe ir seguido

de un espacio en blanco.
    - Cadenas vacías y espacios en blanco son convertidos a cero.
- En caso de no poder convertir el valor pasado a la función *Number*, se debe lanzar un error indicando fila y columna de la instrucción y la razón por la cuál no es posible realizar la conversión.

```
console.log(BigInt(" -2 ")); // -2n

console.log(BigInt("  ")); // 0n
console.log(BigInt(true)); // 1n
console.log(BigInt(true && false)); // 0n

let a = 100;
let b = BigInt(a) / 100n;
console.log(b);// 1n
```

Función Boolean

Valores de otros tipos pueden convertirse a tipo *boolean*. La función *Boolean()* se puede resumir de la siguiente manera:

- Valores u operaciones booleanas son retornadas tal cual.
- *undefined* o variables que no tengan un valor definido son retornadas como *false*.
- Los valores 0 y -0 son convertidos a *false*, otros valores de tipo *number* toman el valor de *true*.
- Los valores 0n y -0n son convertidos a *false*, otros valores de tipo *bigint* toman en valor de true.
- Cadenas vacías son convertidas en *false* y el resto de cadenas son convertidas en *true*.

```
let _flag: string;

console.log(Boolean("true")); // true
console.log(Boolean("false")); // true
console.log(Boolean("")); // false
console.log(Boolean(" ")); // true
console.log(Boolean(_flag)); // false
```

Función String

La función *String()*, se puede resumir de la siguiente manera:

- Las cadenas son regresadas tal cual.
- Variables sin valor o el valor *undefined*, es transformada a “undefined”.
- Los valores *true* y *false* son convertidos a *“true”* y *“false”*.
- Los valores de tipo *number*, son convertidos a cadenas y el mismo caso para las variables o valores de tipo *bigint*.

```
const n1 = 10n;
const n2 = 10;
let _f: boolean;
console.log(String(n1)); // 10
console.log(String(n2)); // 10
console.log(String(_f)); // "undefined"

if (String (n1) == String(n2)) {// true
    console.log("n1 and n2 are the same"); // n1 and n2 are the same
}
```

Métodos y propiedades del tipo de dato *string*

length: *number*

Indica el número de unidades de código UTF-16 en la cadena (longitud de la cadena).

charAt(index: number): string

Devuelve el carácter en la posición especificada en la cadena, si *index* es menor que cero o más grande que la longitud de la cadena, se deberá lanzar un error de ejecución: “índice fuera de los límites”, indicando el número de fila y columna donde surge el error.

toLowerCase(): *string*, toUpperCase(): *string*

Devuelve la cadena en minúsculas o en mayúsculas, respectivamente.

concat(str: string): string

Combina el texto de dos cadenas y devuelve una nueva cadena.

```
let s1 = 'Hello';
let s2 = 'World';

const n1 = s1.length;
const n2 = s2.length;
console.log('La suma es:', n1 + n2); // 10

// hello
console.log(s1.toLowerCase());

// WORLD
console.log(s2.toUpperCase());

// HELLO WORLD
console.log(s1.concat(" ").concat(s2).toUpperCase());
```

Instrucción console.log

Su función es imprimir en la consola el valor o valores u operaciones separados por coma que se le pasan como argumentos.

```
let fullname = 'Jose Perez';
let age = 21;

// My name is Jose Perez
console.log("My name is:", fullname);
// My name is Jose Perez and I'm 21 years old
console.log("My name is", fullname, "and I'm", age, "years old");
```

Instrucción if, else if, else

Una instrucción if puede incluir una o más expresiones que devuelven un valor booleano, si el valor booleano es verdadero, se ejecutan el conjunto de sentencias en el cuerpo de la instrucción if.

```
if (true) {
      console.log('Esto siempre se ejecutará');
}

if (false) {
      const a: number = 10, b: bigint = 12n;
      if (a < b) {
            console.log('a < b');
      }
      console.log('esto nunca se ejecutara');
}


const x = 10, y = 20;
let z = 100;
if (x > y) {
      z++;
      console.log('El mayor es: ' + x);
} else if (x < y) {
      z--;
      console.log('El mayor es: ' + y);
} else {
      console.log('Los numeros son iguales');
}
```

Instrucción for Loop

Se utiliza para ejecutar un bloque de código un número determinado de veces, que se especifica mediante una condición.

```
for (primera_expresion; segunda_expresion; tercera_expresion) {
    //  instrucciones para ser ejecutadas repetidas veces
}
```

- Primera expresión: La primera expresión es ejecutada antes de que el ciclo empiece y consiste en una declaración o conjunto de declaraciones separadas por comas o una asignación de alguna variable que ya exista.
- Segunda expresión: La segunda expresión consiste en una condición, valor booleano u operación booleana para que se ejecute el bucle.
- La tercera expresión es ejecutada después de la ejecución de cada bloque de código en el cuerpo del for.

```
for (let i = 0; i < 3; i++) {
  console.log ("Block statement execution no." + i);
}

const num:number = 5;
let i:number;
let factorial = 1;

for(i = num; i>= 1; i--) {
   factorial = factorial * i;
}
console.log(factorial)


for (let i = 0, j = 10; i < j && j > 5; i++) {
      console.log('i: ' + i + ', j: ' + j);
      if (i % 2 == 0) {
            j = j - 2;
      }
}
```

Instrucción While

El ciclo while ejecuta un conjunto de instrucciones cada vez que la condición especificada se evalúa como verdadera. En otras palabras, el ciclo evalúa la condición antes de que se ejecute el bloque de código.

```
while(condition) {
   // statements if the condition is true
}
```

```
let num : number = 5;
let factorial : number = 1;

while(num >= 1) {
   factorial = factorial * num;
   num--;
}
console.log("The factorial  is "+factorial);
```

Instrucción Do While

El ciclo Do While es similar al ciclo While, excepto que la condición se da al final del ciclo. El bucle Do While ejecuta el bloque de código al menos una vez antes de verificar la condición especificada. Para el resto de las iteraciones, ejecuta el bloque de código solo si se cumple la condición especificada.

```
do {
// code block to be executed
}
while (condition expression);
```

```
let i: number = 2;
do {
    console.log("Block statement execution no." + i );
    i++;
} while ( i < 4);
```

Instrucción break

La instrucción break se usa para terminar o tomar el control de un ciclo. El uso de break hace que la ejecución del programa salga del ciclo en el cual se encuentra, es decir que esta instrucción es sólo permitida dentro de los loops o bucles for, while y do-while. Su sintaxis es la siguiente:

```
let i : number = 1;
while(i <= 10) {
   if (i % 5 == 0) {
      console.log ("The first multiple of 5  between 1 and 10 is : "+i);
      break;     //exit the loop if the first multiple is found
   }
   i++
}  //outputs 5 and exits the loop
```

Instrucción continue

La instrucción ***continue*** salta las instrucciones subsiguientes en la iteración o ciclo en el que se encuentra y lleva el control de la ejecución al comienzo del ciclo. A diferencia de la sentencia ***break***, ***continue*** no sale del ciclo, sino que termina la interacción actual y comienza la iteración subsiguiente. Es importante aclarar que esta sentencia es permitida únicamente dentro de los loops for, while y do-while.

```
let num:number = 0
let count:number = 0;

for(num=0;num<=20;num++) {
   if (num % 2==0) {
      continue;
   }
   count++;
}
console.log (" The count of odd values between 0 and 20 is: "+count);
//output is 10
```

Es importante aclarar que ninguna instrucción puede venir después de la instrucción ***break*** o la instrucción ***continue***, en caso contrario se debe de indicar un error semántico, con el listado de instrucciones que no se ejecutarán, incluyendo la fila y columna de las mismas.

Funciones

Las funciones son los componentes básicos del código legible, mantenible y reutilizable. Una función es un conjunto de declaraciones para realizar una tarea en específico. Las funciones organizan el programa en bloques lógicos de código. Una vez definidas, las funciones pueden llamarse para acceder al código, haciendo el código reutilizable. La definición de una función le indica al compilador el nombre de la función, el tipo de dato que devuelve la función y los parámetros que recibe la función.

Ejemplo de la declaración de funciones:

```
function sayHello(): void {
      console.log('Hello there!');
}

function sayHello2(name: string): void {
      const greeting = 'Hello ' + name + '!';
      console.log(greeting);
}
```

Nótese que las funciones declaradas arriba no *devuelven* o *retornan* ningún valor, por tanto se ha usado el keyword *void*. También nótese que la función *sayHello2*, recibe un parámetro llamada *name* de tipo *string*, en general una función puede o no recibir parámetros, y en caso de recibir más de un parámetro estos van separados por comas y siguen la sintaxis:

```
function function_name (param1: type, param2: type, param3: type)
```

Una función debe ser llamada para ejecutarla, este proceso se denomina invocación de una función:

```
function test() {   // function definition
   console.log("function called")
}

test();             // function invocation
```

Instrucción return

Las funciones también pueden devolver un valor junto con el control, devuelta a donde se ha llamado la función. La sintaxis es la siguiente:

```
function function_name():return_type {
   //statements
   return value;
}
```

- *return_type*, puede ser cualquier tipo de dato válido (boolean, bigint, number, string).
- Una función de retorno debe de tener o finalizar con una instrucción *return*.
- No puede haber más instrucciones después de una instrucción *return*, en caso de haber más instrucciones, se deberá de mostrar un error semántico indicando que instrucciones no se ejecutarán por estar debajo de una instrucción *return*.
- El tipo de dato del valor a retornar debe coincidir con el tipo de dato de la función.
- Es posible omitir el tipo de dato que la función retorne, en tal caso el compilador deberá inferir que tipo de dato devuelve la función y también deberá de validar que todas las posibles instrucciones *return* dentro de la función devuelvan el mismo tipo de dato.

```
 //function defined
function greet():string { //the function returns a string
   return "Hello World";
}

function caller() {
   const msg = greet() //function greet() invoked
   console.log(msg)
}

//invoke function
caller();
```

- Se declara una función *greet()*, el valor de retorno es de tipo *string*.
- La función *caller()*, invoca a la función *greet()*, y está última devuelve la cadena *"Hello World"*, que se guarda en la variable msg y posteriormente se imprime en la salida estándar.
- La función *caller()* es invocada y está ejecuta el proceso descrito en los incisos anteriores.

Los parámetros de una función son un mecanismo para pasar valores a una función. Estos forman parte de la firma de una función. Los valores de los parámetros se pasan a la función

durante su invocación y la cantidad y tipo de parámetros especificados en la definición de la función deben coincidir con los valores que se le pasan durante su invocación.

Mientras se invoca a una función, en general hay dos formas en que los argumentos se pueden pasar a una función: Por valor y por referencia. Pero para este caso se usará el ***paso por valor***. Es decir se copiará el valor de los argumentos pasadas a la función, y los cambios realizados en el parámetro dentro del cuerpo de la función no afectarán al argumento.

Recursividad

La recursividad es una técnica para iterar sobre una operación haciendo que una función se llame a sí misma repetidamente hasta que llegue a un resultado. La recursividad se aplica mejor cuando necesita llamar a la misma función repetidamente con diferentes parámetros dentro de un bucle.

```
function factorial(n: number) {
   if (n <= 0) {         // termination case
      return 1;
   } else {
      return (number * factorial(n - 1));     // function invokes itself
   }
}
console.log(factorial(6));      // outputs 720
```

Para el ejemplo anterior, nótese que no se indico el tipo de valor a retornar, pero el compilador infiere que retornará un tipo de dato *number*, también el compilador debe de verificar que todos los caminos posibles devuelvan un valor.

Para la función anterior, si el cuerpo de la instrucción *if* nunca se ejecuta, debido a la condición, el cuerpo de la instrucción *else* siempre se ejecutará y por tanto siempre hay un valor a retornar.

Importante:

- Cada función puede acceder a las variables que están definidas antes de la declaración de la función, pero no puede acceder a las variables que estén definidas después de la declaración de la función.
- Cada función puede invocar a las funciones definidas sin importar si están antes o después de la función invocadora.

Propiedades matemáticas estáticas presentes en TypeSecure

Math.E: *number*

Devuelve el número Euler que es la base de los logaritmos naturales, aproximadamente 2.718

Math.PI: *number*

Devuelve el número PI, aproximadamente 3.141592

Math.SQRT2: *number*

Raíz cuadrada del número 2, aproximadamente 1.414

Funciones estáticas presentes en TypeSecure

Math.abs(value: number): number

Valor absoluto del valor pasado como argumento.

Math.ceil(value: number): number

Devuelve el entero más pequeño mayor o igual que *value*.

Math.cos(value: number): number

Devuelve el coseno de value en radianes.

Math.sin(value: number): number

Devuelve el seno de value en radianes.

Math.tan(value: number): number

Devuelve la tangente de value en radianes.

Math.exp(value: number): number

Devuelve “*e* elevado a *value”*, donde *value* es el argumento y *e* es el número de Euler (2,718…, la base del logaritmo natural).

Math.floor(value: number): *number*

Devuelve el entero más grande menor o igual que *value*.

Math.pow(base: number, potencia: number): number

El método estático Math.pow() devuelve el valor de la *base* elevada a la *potencia*.

Math.sqrt(value: number): *number*

Devuelve la raíz cuadrada de *value*.

Math.random(): *number*

El método estático *Math.random()* devuelve un número *pseudoaleatorio* de punto flotante que es mayor o igual que 0 y menor que 1.

```
function degToRad(degrees: number) {
    return degrees * (Math.PI / 180);
}

function radToDeg(rad: number) {
    return rad / (Math.PI / 180);
}

const deg = 45; // 45 grados
const rad = degToRad(deg);
const radius = 10;
const area = (rad / 2) * Math.pow(radius, 2);

// El area es 39n
console.log("El area es:", BigInt(Math.floor(area)));

if (area < Math.SQRT2 * Math.PI) {
    console.log('The circle sector is too small');
} else {
    console.log('The size is ok');
}
```

Comentarios

Existen comentarios de línea y comentarios multilínea:

```
// comentarios de linea

/*
    Comentarios multilinea
*/

/* otro comentario multilinea */
```

Función printAst(function_name: string) : *void*

La función *printAst()* recibirá el nombre de una función e imprimirá o generará una imagen en formato png o similar, con el ast del cuerpo de dicha función.

Función getSymbolTable(): *void*

La función *getSymbolTable()* generará una archivo en formato html con la tabla de símbolos en el contexto en donde se invoque la función, se deberá incluir el nombre de cada variable o función a la cual la tabla de símbolos tiene acceso, así como su valor actual o tipo de retorno y la línea y columna donde fue declarada la variable o función.

Reportes

Reporte de errores léxicos y sintácticos

Generar reporte en formato html de errores léxicos y sintácticos indicando número de línea y columna del error y una descripción del mismo.

Reporte de errores semánticos

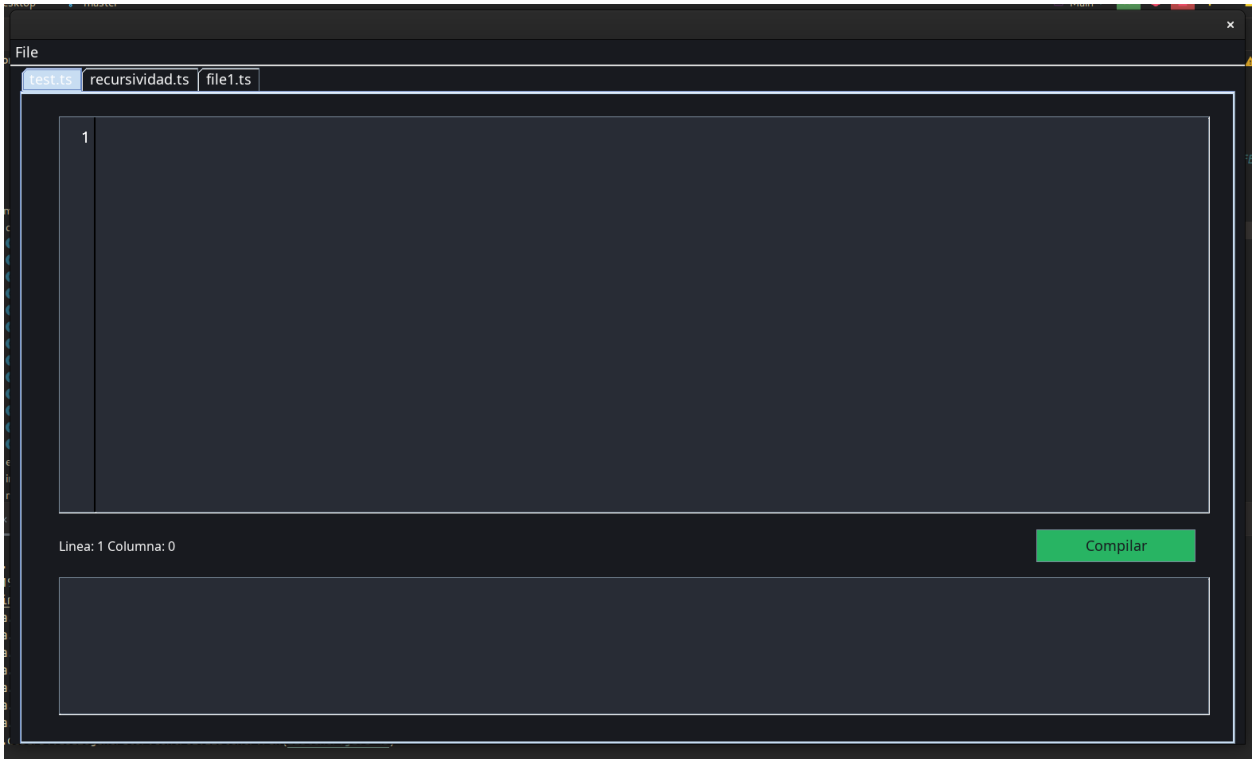
Generar un reporte en formato html de los posibles errores semánticos, indicando número de línea y columna del error y una descripción del mismo.

Sobre la Interfaz gráfica

La aplicación debe tener interfaz gráfica desarrollada con Java Swing o algún similar, tendrá un área para edición de archivos de texto plano (o en formato .ts) además de las opciones de:

- Abrir un nuevo archivo.
- Guardar archivo.
- Guardar como.
- Cerrar archivo.

Y tendrá la posibilidad de abrir varios archivos a la vez, en forma de pestañas, pero se ejecutará siempre el archivo que se esté visualizando. Además tendrá un área para visualizar la salida producida por algún error de ejecución o por la instrucción *console.log*.



Consideraciones del proyecto:

- La aplicación debe ser desarrollada para plataforma desktop usando Java como lenguaje de programación.
- Usar las herramientas JFlex y CUP para todo tipo de análisis léxico y sintáctico.
- Las copias obtendrán una nota de cero y se notificará a coordinación.
- Es posible usar código de internet siempre y cuando se entienda la funcionalidad del mismo para que se tome como válido.

Entrega:

- Fecha de entrega: Viernes 19 de mayo (23:59 horas).
- Componentes a entregar:
    - Código fuente (repositorio github).
    - Manual técnico, detallando la organización del proyecto, descripción de las expresiones regulares y gramática utilizada.
    - Manual de usuario.

Calificación:

Pendiente de definir.