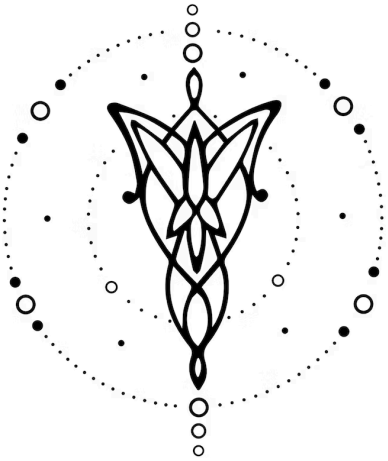


# Especificación

## Transpiler de Subset de TypeScript a JavaScript

v1.0.0



# 1. Equipo



| Nombre | Apellido | Legajo | E-mail |
|---|---|---|---|
| Timoteo | Feeney | 62742 | tfeeney@itba.edu.ar |
| Agustin | Alonso | 63316 | agusalonso@itba.edu.ar |
| Magdalena | Cullen | 63065 | mcullen@itba.edu.ar |
| Tomás | Becerra | 63732 | tobecerra@itba.edu.ar |



# 2. Repositorio

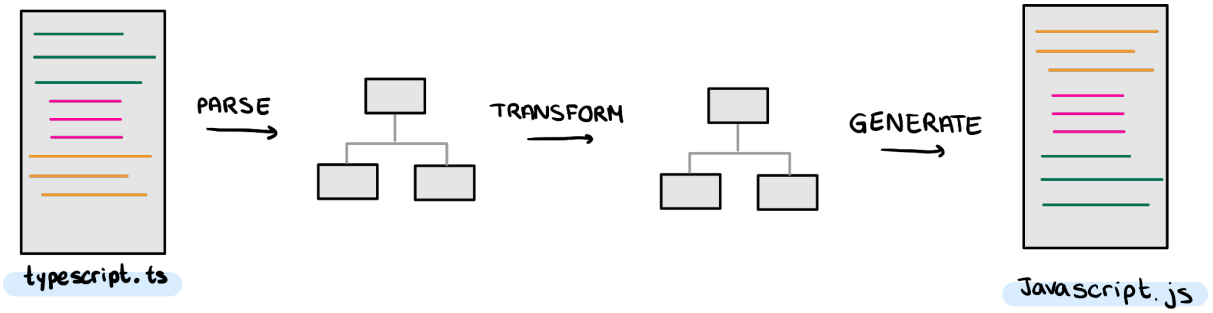
La solución y su documentación serán versionadas en: [TypeScript-JavaScript-Compiler](TypeScript-JavaScript-Compiler).

# 3. Dominio

Desarrollar un transpiler que toma como entrada un programa escrito en TypeScript, de un subset en específico, lo valide, y devuelva o genere como salida el código equivalente en JavaScript. A partir de este proyecto, pretendemos poder aprender sobre la sintaxis y características de TypeScript, además de la traducción de un lenguaje con utilización de tipos (TypeScript) a uno sin estas (JavaScript).

El mayor desafío de este proyecto, creemos que consistirá en poder traducir de manera correcta el sistema de tipos que utiliza TypeScript, los cuales proporcionan una validación más estricta a diferencia de JavaScript, el cual carece de este sistema. El transpiler no solo debería ser capaz de verificar la sintaxis del programa de entrada, si no que a su vez debería poder garantizar que el código de salida generado sea funcional y respete la lógica original de TypeScript.

A través de este proyecto, podremos familiarizarnos y adentrarnos más en los dos lenguajes, tanto TypeScript como JavaScript, y aprenderemos las diferencias fundamentales entre ambos, además de la experimentación con herramientas que facilitarán la transición y compatibilidad entre ellos.




# 4. Construcciones

El transpiler desarrollado tomará un script programado en lenguaje TypeScript y lo traducirá a JavaScript, lo cual es interesante para tratar un lenguaje con tipado estático a un dinámico. Una parte clave para este proyecto será lograr garantizar que el código mantenga la lógica y funcionalidad original. Por lo tanto el lenguaje desarrollado debería ofrecer las siguientes construcciones, prestaciones y funcionalidades:

(I). **Análisis léxico:** el lexer divide el código TypeScript en tokens para representar las unidades mínimas del lenguaje, como identificadores, operadores, palabras claves, etc.

(II). **Análisis sintáctico:** el parser estructura esos tokens generados por el lexer para construir un árbol sintáctico abstracto(AST) que representa la estructura lógica/jerárquica del programa

(III). **Traducción del sistema de tipos:** TypeScript tiene un sistema de tipos que ayuda a garantizar y validar la coherencia del código mientras que JavaScript no lo tiene. Por lo tanto el mayor desafío será pasarlo a no tipado. Nuestro transpiler tendrá que eliminar el sistema de tipos pero tendrá que validar igual la traducción de código sea correcta, es decir, deberá asegurarse de que las funciones y objetos traducidos respeten su estructura lógica.

(IV). **Validación y Testing:** luego de la traducción el transpiler deberá realizar un serie de pruebas para asegurar que el código generado es correcto y funcional en relación al código original de typescript.

# 5. Casos de Prueba

Se proponen los siguientes casos iniciales de prueba de **`aceptación`**:

(I). Un programa que convierta correctamente tipos básicos de TypeScript a su equivalente en JavaScript. Por ejemplo, los tipos (*number*, *string*, *boolean*) se deberían eliminar en la conversión, ya que JavaScript no requiere anotaciones de tipo.

(II). Un programa que convierta adecuadamente las funciones con tipos de entrada y salida. Las anotaciones de tipo en los parámetros y el tipo de retorno se deberían eliminar, y la función se debería convertir en JavaScript estándar. (Eliminando los tipos, pero manteniendo la lógica correcta).

(III). Un programa para verificar que las clases sean convertidas correctamente, sin tipos, manteniendo la funcionalidad. Es decir, manteniendo su rol en JavaScript, pero las anotaciones de tipo se eliminarían.

(IV). Un programa donde las interfaces de tipos de datos en TypeScript sean eliminadas en el código JavaScript resultante, pero conservando la estructura de datos.

(V). Un programa donde los arrays y objetos sean convertidos correctamente, eliminando los tipos pero manteniendo la estructura y los valores.

(VI). Un programa que convierta las funciones *async* a funciones que devuelven promesas *Promise.resolve()*, y las anotaciones de tipo se eliminen.

(VII). Un programa que use funciones o clases genéricas en typescript que el transpiler elimine el parámetro de tipo genérico y deje la funcionalidad.

(VIII). Un programa que use tipos unión en TypeScript, se eliminen los tipos pero conserve la lógica.

(IX). Un programa que use tipos opcionales en TypeScript, se elimine la marca opcional y de tipos pero mantener la funcionalidad.

(X). Un programa que define un enum se convierte correctamente en un objeto en JavaScript, y la función debe seguir siendo válida.

Además, los siguientes casos de prueba de **`rechazo`**:

(I). Un programa que contenga un error sintáctico, como una declaración incompleta de una función o variable.

(II). Un programa que use tipos incorrectos, es decir que, por ejemplo intente asignar un string a una variable tipada como number.

(III). Un programa que busque realizar una operación inválida, como restar un string y un number.

(IV). Un programa que busque usar una variable no declarada o una variable fuera de su scope.

(V). Un programa tenga una declaración duplicada de una misma variable en un mismo ámbito.

# 6. Ejemplos

Programa bien formado de TypeScript, en donde se utilizan elementos como funciones, ciclos, operadores y asignaciones de variables. Cuando se define una nueva variable, debemos soportar dos posibles escenarios: que se defina explícitamente el tipo, o en su defecto que se lo deje sin asignar para que TypeScript lo deduzca.

| TypeScript | JavaScript |
|---|---|
| `// Declaración de variable especificando el tipo:`<br>`const n: number = 10;`<br><br>`console.log(sumNumbers(n));`<br><br>`// Funcion`<br>`function sumNumbers(n: number): number`<br>`{`<br>`  // Deducción de tipos`<br>`  let count = 0;`<br><br>`  // Ciclos`<br>`  while (n > 0) {`<br>`  # Operaciones`<br>`  count += n;`<br>`  n--;`<br>`  }`<br><br>`  return count;`<br>`}` | `// Eliminación del tipado`<br>`const n = 10;`<br><br>`console.log(sumNumbers(n));`<br><br>`// Función sin tipado`<br>`function sumNumbers(n) {`<br>`  let count = 0;`<br><br>`  // Ciclos`<br>`  while (n > 0) {`<br>`  // Operaciones`<br>`  count += n;`<br>`  n--;`<br>`  }`<br><br>`  return count;`<br>`}` |

Programa que define una interfaz como estructura de datos, garantizando que cualquier objeto que se instancie a partir de dicha interfaz, tenga las propiedades “name”, “age” y opcionalmente “job”.

| TypeScript | JavaScript |
|---|---|
| `// Definición de una interfaz`<br>`interface Person {`<br>`  name: string;`<br>`  age: number;`<br>`  job?: string;`<br>`}` | `// Definición de una interfaz no se hace en JavaScript`<br><br>`// Utilizar el objeto`<br>`const person = {`<br>`  name: 'Alice',`<br>`  age: 30,` |

```
// Utilizar dicha interfaz
const person: Person = {
    name: 'Alice',
    age: 30,
    job: 'Developer'
};

// Imprimir el estado de la persona
console.log(person);
```

```
    job: 'Developer'
};

// Imprimir el estado de la persona
console.log(person);
```