ChatGPT

# Guía Técnica: Fase de Compilación – Análisis Semántico de Compiscript

## 1. Configuración del Entorno de Desarrollo (Python + ANTLR4)

Antes de codificar, prepara el entorno en Python con ANTLR4:

- **Python 3:** Asegúrate de tener instalado Python 3 en tu sistema.
- **Instalar ANTLR4:** Puedes usar **pip** para instalar tanto la herramienta de generación (`antlr4`) como el *runtime* de ANTLR para Python [1]. Por ejemplo:

  ```
  pip install antlr4-tools antlr4-python3-runtime
  ```

  Esto proporciona el comando `antlr4` en tu terminal y la librería runtime de ANTLR para Python. Alternativamente, descarga el JAR de ANTLR4 (por ejemplo, `antlr-4.13-complete.jar`) y ejecútalo con Java cuando generes el parser.
- **Docker (opcional):** Si dispones de la imagen Docker proporcionada en el material de Compiscript, puedes usarla en lugar de una instalación local [2]. El contenedor Docker ya incluye ANTLR y un entorno configurado; solo debes montar tu proyecto dentro del contenedor.

Con el entorno listo, coloca el archivo de gramática `Compiscript.g4` en tu proyecto. Esta gramática define la sintaxis del lenguaje Compiscript (un subconjunto de TypeScript) y será la base para generar el lexer y parser. Asegúrate de tener también el *runtime* de ANTLR4 para Python instalado (como hicimos arriba) para poder usar en código las clases generadas [1].

## 2. Generación del Lexer y Parser con ANTLR

Una vez configurado ANTLR, compila la gramática **Compiscript.g4** para generar el lexer y parser en Python. Desde la terminal, en el directorio donde esté `Compiscript.g4`, ejecuta el comando de ANTLR4 con las opciones apropiadas:

```
antlr4 -Dlanguage=Python3 -visitor -no-listener Compiscript.g4
```

Este comando indica a ANTLR que genere código en Python 3 (`-Dlanguage=Python3` [3]), creando las clases base del patrón Visitor (`-visitor`) y omitiendo los *listeners* (`-no-listener`) que no usaremos [4]. Al ejecutarlo, ANTLR leerá la gramática y producirá varios archivos `.py` en el mismo directorio (a menos que especifiques un directorio de salida con `-o`). Entre los archivos generados típicos estarán:

- `CompiscriptLexer.py` – Analizador léxico (convierte la secuencia de caracteres de entrada en tokens).

- `CompiscriptParser.py` – Analizador sintáctico (contiene las reglas de parsing según la gramática).
- `CompiscriptParserVisitor.py` – Clase base *visitor* con implementación vacía (y posiblemente `CompiscriptParserVisitor` como interfaz o clase para heredar).

**Nota:** Si estás usando Docker, dentro del contenedor ya puedes ejecutar directamente el comando anterior para generar los archivos [5]. Tras la generación, verifica que los archivos `.py` aparezcan. Si usaste el JAR de ANTLR manualmente, el comando sería similar reemplazando `antlr4` por `java -jar antlr-4.13-complete.jar`.

## 3. Construcción y Visualización del Árbol Sintáctico

Con el parser generado, puedes probar que el análisis sintáctico funciona y obtener el **árbol sintáctico (parse tree)** de un programa de ejemplo. Esto ayuda a validar la gramática y servirá de base para el análisis semántico. Crea un archivo de prueba, por ejemplo `ejemplo.cps`, con código Compiscript válido. Luego, escribe un pequeño script en Python (o utiliza el `Driver.py` proporcionado) para *parsear* el archivo y mostrar el árbol:

```
from antlr4 import FileStream, CommonTokenStream
from CompiscriptLexer import CompiscriptLexer
from CompiscriptParser import CompiscriptParser

# Cargar el archivo fuente .cps
input_stream = FileStream("ejemplo.cps", encoding="utf-8")
lexer = CompiscriptLexer(input_stream)
tokens = CommonTokenStream(lexer)
parser = CompiscriptParser(tokens)

# Parsear utilizando la regla inicial de la gramática (asumimos se llama
'program')
tree = parser.program()
print(tree.toStringTree(recog=parser))
```

Al ejecutar lo anterior, si la gramática está bien definida, `toStringTree` imprimirá la estructura jerárquica del árbol sintáctico en notación de paréntesis anidados [6]. Por ejemplo, podrías ver una salida del estilo:

```
(program (declaration ...) (functionDecl ...) ...)
```

que refleja la estructura del código de entrada. Esto confirma que el lexer y parser están reconociendo correctamente el lenguaje.

Puedes visualizar el árbol de forma más amigable usando herramientas como **ANTLR4 GUI Test Rig** o exportándolo a formatos como Graphviz, pero para efectos del proyecto, el string estructurado es suficiente. Asegúrate de que un programa sintácticamente correcto no genere errores al parsear (en el driver por defecto, si no hay errores de sintaxis, no se imprime nada [7]). Si hay errores, ANTLR los

reportará en la consola con línea y posición, lo cual indica que debes corregir la gramática o el input de prueba.

## 4. AST Simplificado (Abstract Syntax Tree) – Diseño Opcional

Además del *parse tree* completo, es común en compiladores construir un **AST (Árbol de Sintaxis Abstracta)** simplificado. El AST elimina nodos sintácticos que no aportan a la semántica (por ejemplo, símbolos de puntuación, nodos intermedios artificiales) y representa la estructura lógica del programa. En este proyecto no es obligatorio crear un AST propio, pues puedes operar directamente con el árbol de ANTLR; sin embargo, es **recomendado** para mayor flexibilidad [8] [9].

**¿Cómo diseñar el AST?** Puedes definir clases Python que representen los diferentes tipos de nodos: por ejemplo `ASTNode` como clase base, y subclases como `BinaryOpNode`, `NumberNode`, `VarRefNode`, `FunctionCallNode`, etc. Cada nodo contendrá los campos relevantes (p. ej., un `BinaryOpNode` tendría el operador y referencias a los nodos hijo izquierdo y derecho). Un enfoque es construir el AST durante el recorrido semántico: en cada método `visit` del visitor, crear y retornar objetos de estas clases en lugar de usar directamente el parse tree [8].

Por ejemplo, podríamos definir algunas clases simples para el AST:

```
class ASTNode:
    pass

class BinaryOpNode(ASTNode):
    def __init__(self, op: str, left: ASTNode, right: ASTNode):
        self.op = op
        self.left = left
        self.right = right

class NumberNode(ASTNode):
    def __init__(self, value: int):
        self.value = value
```

Luego, en el visitor (ver siguiente sección) al visitar una expresión de suma/resta podríamos crear un `BinaryOpNode` en vez de solo calcular un tipo. **Importante:** Si decides implementar el AST, asegúrate de que **todos** los nodos relevantes queden cubiertos (expresiones, sentencias, etc.), y que el AST conserve la información necesaria para las fases siguientes. Si prefieres no hacerlo, puedes trabajar directamente con los contextos del parse tree de ANTLR en la fase semántica (lo cual es válido en este proyecto [9]).

## 5. Implementación del Visitor de ANTLR para Reglas Semánticas

El corazón de esta fase es el **visitor** que recorre el árbol sintáctico y realiza la **validación semántica**. ANTLR generó una clase base `CompiscriptParserVisitor` (vacía) debido a la opción `-visitor`. Debemos crear una subclase (por ejemplo `SemanticAnalyzer`) que extienda esa base e implemente los métodos

`visitX(ctx)` para las reglas relevantes de la gramática. Cada método de visitor representará la lógica semántica de cierta construcción del lenguaje.

A continuación, se detallan las reglas semánticas por **categoría**, que el visitor debe verificar:

- **Sistema de Tipos (operadores y tipos básicos):** Las operaciones *aritméticas* (`+`, `-`, `*`, `/`, `%`) solo se permiten entre operandos numéricos (p. ej., `integer` o `float`); no se pueden aplicar a booleanos, strings u otros [10]. Si un operando es `integer` y otro `float`, se puede aplicar una *coerción implícita* promoviendo el entero a float para la operación (según la especificación, asumimos promoción de `int` a `float`) [11]. Las operaciones *lógicas* (`&&`, `||` y `!`) solo aceptan operandos booleanos y producen un booleano [12]; cualquier uso de `&&/||` con no-booleanos es error. Las *comparaciones relacionales* (`<`, `<=`, `>`, `>=`) requieren también operandos numéricos y resultan en booleano [13]. Para la igualdad/desigualdad (`==` / `!=`), los dos operandos deben ser del **mismo tipo o tipos compatibles**; por ejemplo, comparar dos enteros o dos booleanos es válido, pero comparar un entero con un string debe ser error [14] [15]. En las *asignaciones* (`=`), el tipo del valor del lado derecho debe ser **asignable** al tipo declarado de la variable del lado izquierdo [16]. Esto significa que debe ser el mismo tipo o convertible sin pérdida: e.g., se puede asignar un `int` a un `float` (promoción automática) pero no un `float` a un `int` sin cast explícito [17] [18]. Además, si una variable no ha sido declarada antes, asignarle algo debe reportar error (ver **Ámbitos** abajo). En el caso de **constantes** (`const`), deben ser inicializadas en su declaración y no pueden ser modificadas posteriormente [19]. Cualquier intento de asignar a una constante después de su definición debe producir un error semántico [20].

- **Ámbitos (scope) y Resolución de Nombres:** El lenguaje soporta variables globales y locales en funciones o bloques anidados [21], por lo que se debe manejar una jerarquía de ámbitos. Cada vez que se declara un identificador (variable, parámetro, función, clase, etc.), se inserta en la tabla de símbolos del ámbito actual. Si se intenta declarar un nombre que **ya existe en el mismo ámbito**, se reporta error de redeclaración [22]. Por otro lado, es válido que un mismo nombre exista en distintos niveles (*shadowing*); e.g., una variable local puede llamarse igual que una global, ocultándola dentro de su bloque [23]. Al usar un identificador (p. ej., en una expresión), el visitor debe buscarlo en la tabla de símbolos comenzando por el ámbito actual y ascendiendo a los padres hasta encontrarlo [24]. Si no se encuentra en ningún nivel, es un error de variable no declarada [25]. Los ámbitos nuevos se crean al entrar en ciertos bloques: por ejemplo, al iniciar el cuerpo de una función, el cuerpo de una clase, o un bloque `{ ... }` dentro de una función, se abre un nuevo entorno local; al salir, se cierra (se regresa al entorno padre) [26] [27]. El visitor debe manejar esto: al entrar a una función o bloque, crear un nuevo scope; al salir (al terminar de visitar ese nodo), descartar el scope. También aplica para funciones anidadas (*closures*), las cuales Compiscript soporta [28] (cada función interna tiene su propio ámbito enclavado dentro de la externa).

- **Funciones:** Al declarar una función, se debe especificar su tipo de retorno (o `void` si no retorna nada) y los tipos de sus parámetros [29]. Semánticamente, se debe insertar la función en la tabla de símbolos global con su firma (nombre, tipos de parámetros, tipo de retorno) para que pueda ser resuelta en llamadas [30]. No puede haber dos funciones con el mismo nombre en el mismo ámbito (asumimos que no hay sobrecarga de funciones en Compiscript) – eso sería una redeclaración inválida. Al visitar la declaración de una función, el visitor debería crear un nuevo ámbito para el cuerpo de la función e **insertar en él los parámetros** como variables locales [31]. Dentro del cuerpo, se debe verificar que **todas las rutas de ejecución de la función no-void retornan un valor del**

**tipo correcto**. Cada sentencia `return` debe llevar una expresión de tipo compatible con el tipo declarado de la función (o carecer de expresión si la función es void) [32] [33]. Ejemplos: en una función declarada `: integer`, un `return;` (sin valor) es error, al igual que `return "texto";` si se espera un entero. En una función `: void`, cualquier `return expr;` con valor es error. Además, se podría advertir si una función no-void podría terminar sin ejecutar ningún `return` (código muerto por falta de return, aunque esto es más complejo de validar estáticamente) [34]. En las **llamadas a función**, el visitor debe verificar: (1) que la función exista (haya sido declarada) [35]; (2) que el número de argumentos en la llamada coincida con el número de parámetros definidos [36]; (3) que el tipo de cada argumento sea compatible (asignable) con el tipo del parámetro correspondiente [37]. Si alguna comprobación falla, se agrega un error semántico, indicando por ejemplo *"Número de argumentos incorrecto"* o *"Tipo de argumento X incompatible"*. Las funciones deben poder llamarse a sí mismas (recursividad) si están declaradas; el diseño de la tabla de símbolos debe permitir que dentro del cuerpo de `f` se reconozca una llamada a `f` (por lo general, insertando la función en la tabla *antes* de procesar el cuerpo). En cuanto a **closures** (funciones definidas dentro de funciones), ya cubrimos la creación de sub-ámbitos; adicionalmente, podrías permitir que la función interna acceda a variables del ámbito externo (si el lenguaje lo permite), aunque en esta fase podríamos no profundizar en eso. Lo importante es manejar correctamente los scopes anidados para que los nombres estén resueltos según alcance léxico.

- **Control de Flujo:** Verifica las reglas semánticas de sentencias de control como `if/else`, bucles (`while`, `do-while`, `for`, `foreach`), y sentencias de salto (`break`, `continue`, `return`). En las condiciones de un `if` o de un loop `while/for`, la expresión debe ser de tipo booleano [38]. Si la condición es numérica u otro tipo no booleano, produce error (Compiscript es de tipado estricto, no convierte implícitamente 0/1 a booleano como C). Dentro de un bucle, las sentencias `break` y `continue` son válidas; pero si el visitor encuentra un `break` o `continue` **fuera** de un contexto de bucle, debe marcar error semántico [39] (ejemplo: un `break` en medio de un bloque normal es inválido). Maneja esto llevando un indicador (p.ej., un contador) en el visitor de si estás dentro de un loop; incrementarlo al entrar a un loop y decrementarlo al salir, así el `visitBreak` puede chequear si `inLoop == 0` para error [40] [41]. La sentencia `return` ya fue discutida en Funciones; solo añadir que si un `return` aparece en el ámbito global (fuera de cualquier función), es un error ya que no tiene efecto válido [42]. En Compiscript también existe `switch/case` [43]; sus reglas incluyen que la expresión del `switch` y los valores de los `case` sean de tipos comparables, y que no haya valores de case duplicados. Además, se recomienda tener un `default` para cubrir todos los casos. Una regla semántica típica es exigir que **cada** `case` **dentro de un** `switch` **termine con** `break` para evitar *fall-through*, aunque esto puede depender de la especificación del lenguaje (no detallada aquí). Asimismo, un `continue` dentro de un `switch` fuera de bucle sería inválido. (Si la implementación de `switch` no es requerida en detalle, puedes omitir esta parte).

- **Clases y Objetos:** Si el lenguaje incluye POO, debemos verificar la semántica de **clases, atributos, métodos,** `this`**, instanciación** y posiblemente **herencia**. En Compiscript se pueden definir clases con atributos (propiedades) y métodos, y crear instancias con `new` [44] [45]. Reglas: Al declarar una clase, regístrala en la tabla de símbolos global (como un símbolo de tipo Clase). Crea también una tabla de símbolos propia para el scope de la clase (donde vivirán los símbolos de sus miembros) [46]. Inserta en ella cada **atributo** con su tipo, comprobando que no haya dos atributos con el mismo nombre en la misma clase (error de redefinición) [47]. Para los **métodos**, aplica las mismas reglas que para funciones: parámetros, tipo de retorno, returns internos, etc., solo que viven dentro del scope de la clase. Puedes representarlos en la tabla de la clase como símbolos de función. Además,

dentro de un método, la palabra clave `this` debe referirse a la instancia actual de la clase. Una buena práctica es insertar un símbolo `this` en el scope del método, de tipo la clase correspondiente, para validar su uso. Por ejemplo, asignaciones a `this.prop` deben verificar que `prop` exista en la clase. En **constructores** (métodos especiales generalmente nombrados `constructor` en Compiscript [48]), se deben tratar como métodos que **no** tienen tipo de retorno explícito (su retorno es implícitamente la instancia). Verifica que las llamadas a `new Clase(...)` correspondan a un constructor definido (en Compiscript, posiblemente cualquier función llamada `constructor` en la clase se invoca). Al instanciar una clase con `new`, comprueba que la clase exista; si requiere parámetros en su constructor, valida que se pasen los argumentos correctos (misma cantidad y tipos) como si fuera una llamada de función [49]. Para el acceso a miembros con el operador `.` (p.ej., `obj.atributo` o `obj.metodo()`), el visitor debe: (1) verificar que el tipo de `obj` es una clase (o instancia de clase) que contiene el miembro solicitado [50]; (2) si el miembro es un atributo, el resultado de `obj.atributo` es el tipo de dicho atributo [51]; si es un método, normalmente deberías verificar la llamada con las reglas de Funciones cuando se invoca. Si el miembro no existe en la clase, es error. Adicionalmente, **herencia**: Compiscript soporta herencia entre clases (`class Perro : Animal {...}`) [52]. Si implementas esto, considera que una clase hija hereda los miembros de la padre. En la tabla de símbolos, la clase hija podría tener un puntero a la `ClassSymbol` de su padre. Reglas típicas: no duplicar atributos heredados en la misma clase (a menos que se permita sobreescritura de métodos), y validar que asignaciones entre objetos de distintas clases respeten la relación de herencia (un objeto de clase hija puede asignarse a una variable de tipo clase padre – *polimorfismo* – pero no al revés sin cast). Si el proyecto no exige manejar herencia, puedes simplificar asumiendo que todos los objetos son exactamente de la clase declarada. En resumen, manejar clases implica gestionar un scope por clase, insertar atributos/métodos, y durante el análisis semántico asegurar instanciaciones y accesos válidos.

- **Listas (Arreglos):** Compiscript incluye listas/arreglos con sintaxis de literales mediante corchetes [53]. Semánticamente, las listas deben ser **homogéneas**: en un literal `[e1, e2, ...]`, todas las expresiones `e1, e2...` deben producir el **mismo tipo** de dato [54]. Si mezclamos tipos (ej. `[1, true]` con entero y booleano), se debe lanzar un error de tipos. Una lista puede tener un tipo genérico representado como, por ejemplo, `List<integer>` o `integer[]`; el parser debería captar esto en la gramática (p. ej., `let x: integer[] = [...]`). Cuando accedemos a un elemento con `lista[idx]`, deben cumplirse dos reglas: (1) que la variable `lista` realmente sea de tipo lista/arreglo; (2) que el índice `idx` sea de tipo `integer` [55]. El resultado de la expresión `lista[idx]` semánticamente tendrá el tipo de elemento de la lista (p. ej., si `lista` es `List<string>`, `lista[idx]` es `string`). Para *asignar* a una posición de la lista (`lista[idx] = expr;`), también validar que `expr` sea del mismo tipo de elemento que la lista almacena [56]; si `lista` es `List<integer>`, no se le puede asignar un booleano en una posición. Otras operaciones de lista (si existen, e.g. `lista.add(expr)`, `lista.size()`) deben comprobar igualmente los tipos esperados: `add` debería aceptar solo elementos del tipo correcto. En la implementación, conviene representar el tipo de lista paramétricamente, por ejemplo con una estructura `Type.LIST(subtype)` que indique el subtipo; así se puede extraer el `subtype` para comparaciones.

- **Reglas Semánticas Generales y Extra:** Además de lo anterior, hay algunas comprobaciones globales que mejoran la corrección del programa: por ejemplo, detectar **código muerto** o inalcanzable. Si tu compilador encuentra, por decir, código después de un `return` al final de una

función, podría lanzar una advertencia de *"código inalcanzable"*. No es un requerimiento estricto detectarlo, pero es una buena práctica mencionarlo. También, cualquier **expresión inválida** que pase el sintáctico pero no tenga sentido semántico debe generar error. Por ejemplo, intentar invocar como función a algo que no es función, o acceder como arreglo a algo que no es lista, ya quedarían cubiertos por las reglas de tipos. Otro caso: el uso de `this` fuera de una clase no tiene significado (podrías reportarlo si sucede, aunque idealmente la gramática ni lo permitiría fuera de métodos de clase). En resumen, el visitor debe actuar como guardián de *todas* las reglas del lenguaje que aseguren que un programa *bien formado* no tenga ambigüedades en tiempo de ejecución. Recomendación: lleva un registro (checklist) de todas las reglas requeridas y asegúrate de implementarlas en algún método del visitor [57] [58].

**Ejemplo de Visitor (Python):** A continuación, mostramos un método simplificado del visitor semántico que maneja la asignación de variables. Esto ilustra cómo integrar las reglas de tipos y el uso de la tabla de símbolos dentro del patrón visitor:

```
from CompiscriptParserVisitor import CompiscriptParserVisitor

class SemanticAnalyzer(CompiscriptParserVisitor):
    def __init__(self):
        self.symbol_table = SymbolTable()  # tabla de símbolos actual (comienza
global)
        self.errors = []

    def visitAssignStatement(self, ctx):
        name = ctx.ID().getText()
        var_type = self.symbol_table.lookup(name)
        expr_type = self.visit(ctx.expr())
        if var_type is None:
            self.errors.append(f"Variable no declarada: {name} (línea
{ctx.start.line})")
        elif not is_assignable(var_type, expr_type):
            self.errors.append(f"Tipos incompatibles: no se puede asignar
{expr_type} a {var_type} (línea {ctx.start.line})")
        return var_type
```

En este fragmento, `symbol_table` representa la tabla de símbolos *actual* (con scope global al inicio). Buscamos el símbolo de la variable asignada; si no existe, agregamos un error. Luego comparamos el tipo de la expresión (obtenido recursivamente con `self.visit(ctx.expr())`) con el tipo de la variable para decidir si es asignable o no, agregando el error correspondiente en caso negativo. Finalmente, retornamos `var_type` como resultado de la visita (muchos métodos `visit` pueden retornar el tipo de la subexpresión evaluada, para usarlo en validaciones en niveles superiores). Métodos similares se implementarían para expresiones aritméticas, lógicas, llamadas de función, etc., siguiendo las reglas descritas arriba (por ejemplo, un `visitAddExpr` sumaría visitar los operandos, obtener sus tipos y verificar que ambos sean numéricos [59] [60], retornando el tipo resultante apropiado).

**Manejo de errores:** Es útil que el visitor acumule una lista de errores (`self.errors`) en lugar de detener la compilación en el primer error, para reportar *todos* los errores semánticos encontrados de una vez. Cada vez que ocurre una violación de regla, agrega un mensaje descriptivo incluyendo la línea (como vimos con `ctx.start.line`). Al final del proceso de visita, si la lista está vacía, significa "compilación semántica exitosa"; si no, se reportan esos mensajes.

## 6. Tabla de Símbolos: Diseño e Implementación de una Pila de Ámbitos

La **tabla de símbolos** es la estructura que almacena información sobre los identificadores declarados (nombre, tipo, alcance, etc.) y soporta las operaciones de insertar y buscar símbolos durante el análisis semántico [61]. Dado que existen múltiples niveles de anidamiento, implementaremos la tabla de símbolos como una **pila de entornos** o tablas enlazadas jerárquicamente (padre-hijo).

Una posible implementación en Python es definir una clase `SymbolTable` que represente un solo ámbito, con un diccionario interno (`symbols`) y una referencia al ámbito padre. Por ejemplo:

```
class SymbolTable:
    def __init__(self, parent=None):
        self.parent = parent      # referencia al ámbito superior
        self.symbols = {}         # diccionario nombre -> símbolo (puede ser
tipo u objeto símbolo)

    def insert(self, name, info):
        if name in self.symbols:
            return False  # ya existe en este scope
        self.symbols[name] = info
        return True

    def lookup(self, name):
        if name in self.symbols:
            return self.symbols[name]
        if self.parent:
            return self.parent.lookup(name)
        return None
```

En este ejemplo, `info` podría ser un objeto que contenga tipo, categoría (var, func, etc.) y atributos extra; o sencillamente el tipo si solo necesitas eso para validar. El método `insert` devuelve False si encuentra una redefinición en el mismo nivel [22] (lo que usaremos para arrojar error de "redeclared symbol"), y `lookup` busca recursivamente en padres si no lo halla en el actual [24].

En la práctica, manejarás una instancia que represente el **scope actual** en el visitor. Cuando entras a un nuevo ámbito (por ej. al iniciar `visitFunctionDecl` o `visitBlock`), creas una nueva `SymbolTable` cuyo padre es el scope actual, y luego reasignas `currentScope` a esa nueva instancia [26]. Al salir del ámbito (al terminar de visitar ese nodo), revierte `currentScope` al padre (p.ej., `currentScope =`

`currentScope.parent`). De esta forma, siempre que hagas `lookup`, primero revisará en el ámbito local y luego ascenderá automáticamente. Este modelo refleja la naturaleza LIFO de los scopes (último en entrar, primero en salir) [26].

Operaciones clave a implementar/en uso en el visitor con la tabla de símbolos:

- **Insertar símbolo:** cuando el visitor procesa una *declaración* (de variable, parámetro, función, clase, etc.), invoca `currentScope.insert(nombre, info)`. Si devuelve False, significa que el nombre ya existía en ese ámbito, por lo que se agrega un error de redeclaración. Si devuelve True, la inserción fue exitosa. Por ejemplo, `visitVarDecl` insertará la variable con su tipo; `visitFunctionDecl` insertará la función en el scope global y luego creará un nuevo scope para sus variables locales [30] [31].
- **Buscar símbolo:** cuando se referencia un identificador (por ej., al visitar una expresión con un `ID`), se hace `info = currentScope.lookup(nombre)`. Si resulta None, es un error de variable no definida [24]. Si encuentra algo, podemos usar esa info (por ej., el tipo) para validaciones. La búsqueda recorre la cadena de padres automáticamente, respetando la *visibility* de las variables (una variable local oculta una global del mismo nombre) [23].
- **Navegar scopes (entrar/salir):** ya descrito, pero en código: podríamos tener métodos utilitarios como `enter_scope()` que haga `currentScope = SymbolTable(parent=currentScope)` y `exit_scope()` que haga `currentScope = currentScope.parent`. Opcionalmente, puedes encapsular esto en la propia clase SymbolTable (e.g., un método `nest()` que devuelva una nueva tabla hija). Lo importante es invocar estos cambios en los puntos correctos del visitor. Por ejemplo, en `visitFunctionDecl`, después de insertar la función en global, haces `enter_scope()` para el cuerpo; al final del cuerpo haces `exit_scope()`. En `visitBlock` (si tu gramática tiene una regla para bloques `{}` anidados), al iniciar el bloque nuevo llamas `enter_scope()`, al salir `exit_scope()`.

Finalmente, define las **clases de símbolo** si lo ves necesario. En implementaciones robustas se crean clases para distintos tipos de símbolo: una clase `Symbol` base, luego `VarSymbol`, `FuncSymbol`, `ClassSymbol`, etc., cada una con campos específicos (tipos, parámetros, miembros…). Por simplicidad, podrías usar strings para tipos y tuplas para funciones (tipo de retorno + lista de tipos de params) al almacenar en la tabla, pero una solución orientada a objetos mejora la claridad. Por ejemplo, `FuncSymbol` podría tener atributos `name, return_type, param_types`. Esto te permite también guardar información adicional si la necesitaras. No olvides que la tabla de símbolos global contendrá también símbolos de clases; una clase podría llevar dentro otra tabla de símbolos para sus propios miembros [62] [63].

Con la estructura de la tabla de símbolos implementada y integrada al visitor, estarás equipado para que durante la visita semántica se puedan validar correctamente cosas como *"variable usada antes de ser declarada"*, *"variable X oculta la global X"*, *"función Y no existe"*, etc., tal como lo requieren los lineamientos del lenguaje.

## 7. Integración del Visitor con la Tabla de Símbolos (y AST) para Reglas Semánticas

Ya tenemos los componentes principales: el parse tree, el visitor semántico y la tabla de símbolos. Ahora, debemos integrarlos de forma coherente:

- **Inicialización:** Al comenzar el análisis semántico (por ejemplo, en una función `analyze(tree)`), crea una instancia de tu `SemanticAnalyzer` (visitor) con una tabla de símbolos global vacía. Inserta en el scope global cualquier símbolo predefinido (si existieran, e.g. funciones nativas o clases nativas). Luego invoca `visitor.visit(tree)` sobre el nodo raíz del parse tree. Esto disparará recursivamente todas las visitas necesarias.

- **Manejo de ámbitos en el visitor:** Como se explicó, el visitor debe actualizar `currentScope` al entrar y salir de ámbitos. Por ejemplo, tu `visitFunctionDecl` hará algo como: insertar la Función en `currentScope` (global), luego setear `currentScope = SymbolTable(parent=currentScope)` para el cuerpo, insertar símbolos de parámetros en este nuevo scope, visitar el cuerpo de la función con `visit(ctx.block())`, y finalmente restaurar `currentScope` al scope global una vez terminado [64] [65]. De manera similar, `visitBlock` (si existe) encapsula su contenido entre un `enter_scope()` y `exit_scope()`. Esta lógica de enter/exit puede implementarse manualmente en cada método visitor pertinente, o usando un *context manager* en Python (with) para automatizar.

- **Uso de la tabla de símbolos para validaciones:** Dentro de los métodos visitor, usarás `lookup` e `insert` según corresponda. Por ejemplo: en `visitVarDecl`, harás `if not currentScope.insert(varName, VarSymbol(type=T, ...)): self.errors.append("Redeclaración de "+varName)` [22]. En `visitAssign` (como vimos en el ejemplo), usas `lookup` para obtener el tipo de la variable y decidir compatibilidad [66] [67]. En `visitId` (si tienes una regla para usar un identificador solo), retornarás el tipo de la variable buscándolo; si `lookup` da None, añades error de no declarada. En `visitFuncCall`, buscas el símbolo de la función (que podrías haber representado con una clase `FuncSymbol` o similar) [35], y de ahí obtienes los tipos esperados para validar los argumentos [36] [37]. Cada regla semántica descrita en la sección 5 se implementa así, combinando la info de la tabla y las estructuras de tipos.

- **De parse tree a AST (si aplica):** Si decidiste construir un AST propio, tu visitor en lugar de devolver tipos (como hicimos en varios ejemplos) podría devolver nodos AST. Por ejemplo, `visitAddExpr` podría crear un `BinaryOpNode(op="+", left= nodoIzq, right= nodoDer)` y retornarlo. En tal caso, la validación de tipos la harías probablemente separada, o incorporando un atributo de tipo en cada nodo AST. Otra forma es tener *dos fases*: un visitor que construye AST y luego otro que hace las comprobaciones recorriendo el AST. Dado el tiempo y alcance, muchos implementarán el análisis semántico directamente sobre el parse tree de ANTLR (retornando tipos en los visit para usar arriba, como en los códigos de ejemplo de la guía [59] [60]). Ambas vías son aceptables. Lo importante es que si construyes AST, **también integres la tabla de símbolos en el proceso** para resolver nombres y adjuntar información necesaria.

- **Acumulación de errores:** Como mencionado, la clase visitor debe tener una estructura (lista) para recolectar errores semánticos. Integra esto en cada check: cuando algo falle, haz

`self.errors.append("mensaje")`. Al terminar `visit(tree)` de la raíz, podrás revisar `visitor.errors`. Por ejemplo, podrías implementar un método en el visitor `getErrors()` que devuelva dicha lista [68]. Si la lista no está vacía, el compilador imprimiría esos errores con sus líneas correspondientes; si está vacía, puedes imprimir un mensaje de "Compilación semántica exitosa" o simplemente nada (como hacía el driver en la fase sintáctica) [69].

En resumen, la integración consiste en que el **visitor** utiliza la **tabla de símbolos** para todas las verificaciones que involucran nombres o tipos declarados, y opcionalmente crea un **AST** para uso futuro. Una vez hecho esto, tu compilador podrá procesar cualquier código Compiscript detectando las violaciones semánticas requeridas. Aprovecha los ejemplos de la sección 5 (muchos vienen de la guía proporcionada) como referencia al implementar cada método del visitor, ajustándolos a Python. Ten presente también los **detalles de la gramática**; por ejemplo, si la gramática llama a la regla de expresiones lógicas `orExpr` / `andExpr`, tus métodos se llamarán `visitOrExpr`, etc., conforme al nombre de las reglas en Compiscript.g4 (usa los nombres generados por ANTLR tal cual).

## 8. Pruebas Unitarias por Categoría de Validación

Para asegurar el correcto funcionamiento del análisis semántico, es fundamental escribir **pruebas unitarias** cubriendo casos válidos e inválidos de cada regla. Puedes usar **PyTest** (con funciones `test_...`) o el módulo **unittest** de Python (clases que heredan de `unittest.TestCase`). La idea es automatizar la verificación de que, por ejemplo, asignar un `float` a un `int` produzca error, o que usar una variable correctamente declarada *no* produzca error [70] [71].

**Configuración de pruebas:** Crea un archivo (o varios) de tests, p.ej. `test_semantic.py`. En cada prueba, prepara un *snippet* de código Compiscript como cadena (o carga un .cps de ejemplo), pásalo por las fases de compilación hasta la semántica, y captura la lista de errores resultante. Puedes implementar una función auxiliar `compile_and_get_errors(code: str) -> List[str]` que haga: lex -> parse -> visitor.visit(tree) -> return errors. De esta manera cada test se reduce a "ejecutar snippet y comprobar errores". Por ejemplo, usando PyTest:

```
from compiler import compile_and_get_errors

def test_asignacion_valida():
    code = "let a: integer; let b: float; b = a;"
    errors = compile_and_get_errors(code)
    assert errors == []  # no debe haber errores (int se asigna a float con
promoción)

def test_asignacion_invalida():
    code = "let a: integer; let b: float; a = b;"
    errors = compile_and_get_errors(code)
    # Debe haber error de tipo incompatible
    assert any("incompatibles" in e for e in errors)
```

En el primer caso esperamos que `errors` esté vacío (asignar int a float es válido con promoción), y en el segundo esperamos al menos un error indicando incompatibilidad de tipos. Este patrón se repite para cada regla:

- **Declaración previa:** prueba usar una variable no declarada. Ejemplo: `x = 5;` sin `let x` antes debería dar error *"Variable no declarada"* [72] . También, declarar dos veces la misma variable en un mismo bloque:

  ```
  code = "{ let x: integer; let x: integer; }"
  ```

  debería producir error de redeclaración. Por contraste, shadowing válido:

  ```
  code = "let x: integer; { let x: integer; }"
  ```

  no debe producir error (la variable interna oculta a la externa) [73] .
- **Operadores aritméticos y lógicos:** probar expresiones bien tipadas vs mal tipadas. Ej: `1 + 2` (válido), `1 + true` (error: bool en suma) [74] . `true && false` (válido), `true && 1` (error: operando no booleano) [74] .
- **Comparaciones:** `3 < 5` (válido si ambos int), `"hola" < "mundo"` debería ser error si no permites comparar strings [75] . `a == b` donde `a` es int y `b` float – dependiendo de si permites conversión; nosotros dijimos tipos iguales estrictamente, así que error a menos que a/b se conviertan.
- **Asignaciones:** ya cubrimos – probar casos de tipos compatibles e incompatibles. Verificar también constante:

  ```
  code = "const PI: integer = 3; PI = 4;"
  ```

  debería dar error de asignación a constante [20] .
- **Listas:** casos como `let L: integer[] = [1,2,3];` (válido), `let M: integer[] = [1, true];` (error por tipos mixtos). Acceso: `L[0] = 5;` (válido), `L[0] = true;` (error tipo) [76] . Índice:

  ```
  code = "let L: integer[] = [1]; let x: string = L[0];"
  ```

  error al asignar int a string (o detectas en el acceso mismatched types).

  ```
  code = "let L: integer[] = [1]; L[\"0\"] = 2;"
  ```

  error porque el índice no es int.
- **Funciones:** definir una función correcta y llamarla bien:

```
code = "function f(a: integer): integer { return a; } let x: integer = f(5);"
```

no errores. Ahora llamadas incorrectas: `f()` con argumentos de menos, o `f(1,2)` con de más, deben dar error de número de argumentos [36]. Llamada con tipo incorrecto: `f(true)` si espera int, error de tipo en argumento [37]. Recursión:

```
code = "function f(n: integer): integer { if(n<=1) return 1; else return n*f(n-1); } let y: integer = f(5);"
```

debería no dar errores (comprueba que reconocer la llamada recursiva funciona). Returns: probar función que olvida retornar algo en alguna ruta, p.ej.

```
code = "function g(): integer { let a: integer = 5; }"
```

– idealmente advertir que no retorna (aunque no sea requerido hacerlo ahora). Probar también retornar mal:

```
code = "function h(): integer { return \"no\"; }"
```

error de tipo de retorno. Y retornar valor en void:

```
code = "function j(): void { return 1; }"
```

error [32].

- **Clases:** si implementaste semántica de clases, prueba: declarar clase con atributo y usarlo:

```
code = "class A { let x: integer; } let a: A = new A(); a.x = 5;"
```

debería ser válido. Ahora acceder atributo inexistente: `a.y = 3;` error *"no tiene un miembro llamado y"* [77]. Invocar método inexistente similar. Herencia (si aplica): crear instancia de subclase y asignarla a variable de superclase (debería ser válido si manejado) – e.g., `class B: A { } let b: B = new B(); let a2: A = b;` (debería permitirse). Asignar superclase a variable de subclase – error si no hay conversión.

- **Control de flujo:** un `if` con condición entera debe dar error [38]; con condición booleana, no. `break` en un loop anidado dentro de otro loop debe ser válido (siempre que esté dentro de *a* loop). Un `break` fuera de cualquier loop:

```
break;
```

error *"break fuera de un ciclo"* [41]. Similar para `continue`. Un `return` en el global:

```
return 5;
```

error porque no está dentro de función.

Organiza estas pruebas por categoría, por ejemplo agrupa en clases o módulos: `test_types.py`, `test_functions.py`, etc., o simplemente separa por secciones dentro del mismo archivo usando comentarios. Lo importante es que cubras **cada regla** mencionada en los requerimientos [78] [79]. Las pruebas deben ser lo más automáticas posible: no dependas de inspección manual de la consola, sino de condiciones `assert` en código. Si utilizas `unittest`, cada regla puede ser un método `test_xyz`. Con PyTest, es similar con funciones libres. Ejecuta estas pruebas con frecuencia durante el desarrollo para detectar regresiones.

Por último, considera configurar un entorno de integración continua (CI) simple si trabajas en equipo (GitHub Actions u otro) que ejecute los tests en cada push, asegurando que no se rompa nada inadvertidamente [80]. Las pruebas unitarias no solo validan tu compilador, sino que también sirven de *documentación ejecutable* de que cada requerimiento semántico está cubierto.

## 9. IDE Educativo Básico (Opcional) – Editor, Compilación y Visualización

Como mejora final, se propone crear un **IDE básico** para Compiscript que permita a los estudiantes escribir código, compilarlo con un botón y ver resultados (errores o árbol) de forma interactiva. *Esto es opcional* pero enriquecedor. Puedes implementar la interfaz en Python usando frameworks como **Tkinter** (incluido en la librería estándar), **PyQt5/PySide** (más moderno, widgets nativos) o incluso una pequeña aplicación web local.

**Características mínimas del IDE** [81]:

- Un panel o área de texto donde el usuario pueda escribir o cargar código fuente (.cps). Idealmente, con **resaltado de sintaxis** para palabras clave, números, strings, etc., aunque esto puede agregarse al final.
- Un botón (p.ej. "Compilar") que al hacer clic ejecute el compilador en memoria: es decir, tome el texto del editor, corra el análisis léxico, sintáctico y semántico usando tus componentes, y recolecte los errores semánticos (y sintácticos si los hay).
- Una manera de **mostrar errores** al usuario, preferiblemente indicando la línea y columna. Por ejemplo, podrías tener un panel abajo o al costado tipo "Consola de errores" listando cada error (e.g., "Línea 3: Variable no declarada: x") [82]. Una mejora es resaltar dentro del propio editor las partes del código con error: por ejemplo, subrayar en rojo la palabra o línea problemática [83]. En Tkinter, el widget `Text` permite agregar *tags* con estilos en rangos de texto; podrías asignar un tag "error" con subrayado rojo y aplicarlo a la porción correspondiente. Necesitas convertir la info de error (línea, columna) a índices del widget. Otra opción más sencilla es resaltar toda la línea en rojo o mostrar un ícono en el margen (esto último es más complejo sin librerías adicionales) [83] [84]. Incluso una simple lista de errores clickable que al hacer clic te lleve a la línea en el editor sería útil [85].

- Opcional: un tab o sección para mostrar el **árbol sintáctico** de manera visual. Esto puede ser desafiante, pero podrías imprimir el `toStringTree` en un TextArea monoespaciado. O dibujar un árbol usando Canvas o una librería gráfica. Dado que es un IDE educativo, con mostrar el árbol en texto estructurado ya se logra el propósito.
- Si implementaste más fases (por ejemplo, generación de código intermedio), podrías añadir pestañas para mostrar ese resultado, pero para análisis semántico nos limitamos a errores y quizá el árbol.

**Elección de tecnología:** Si vas con **Tkinter**, obtienes simplicidad y cero dependencias, aunque la estética es básica. Un `tk.Text` para el editor, un `tk.Button` para compilar, y un `tk.Listbox` o `tk.Text` de solo lectura para errores pueden ser suficientes. Con **PyQt5/PySide**, tienes widgets más sofisticados; por ejemplo, `QTextEdit` para código (que también soporta resaltado por formato de texto) y puedes crear ventanas más elaboradas. La curva de aprendizaje es mayor que Tkinter, pero hay ejemplos en línea de editores de código con PyQt. La vía **web** implicaría usar, por ejemplo, un pequeño servidor Flask para correr el compilador, y una página HTML/JS con un editor embebido (como **Monaco Editor** de VSCode o **Ace**) [86]. Esto ofrece la mejor experiencia de editor, pero añade complejidad de comunicación entre frontend y backend. Dado el objetivo educativo, quizá la ruta más rápida es Tkinter o PyQt en una app de escritorio monolítica (sin cliente/servidor) [87] [86].

**Implementación básica con Tkinter (ejemplo):**

- Crear una ventana principal (`tk.Tk()`), ajustar título y tamaño.
- Añadir un menú opcional (Archivo->Abrir/Guardar, etc.) y el área de texto para código. Para dividir se puede usar `tk.PanedWindow` o simplemente colocar el Text en top.
- Colocar un botón "Compilar" abajo. Al pulsarlo, obtén el contenido del Text (`text_widget.get("1.0", "end")`), ejecútalo por el compilador (llamando a tu función principal que hace lex+parse+semántica) y captura errores.
- Limpiar la vista de errores anterior (quitar highlights de la vez pasada, por ejemplo removiendo tags de error).
- Si no hubo errores, quizás mostrar un diálogo "Compilación exitosa" o simplemente nada. Si sí hubo, para cada error: resaltar la línea en el editor (p.ej., obtener nº de línea del mensaje, luego `text_widget.tag_add("error", f"{lineno}.0", f"{lineno}.end")` en Tkinter) y/o volcar los mensajes en un panel de errores.
- Para resaltado de sintaxis (opcional), se puede usar la técnica de aplicar tags al texto según patrones (palabras clave, etc.) cada vez que se modifica el texto. Esto puede ser complejo de hacer manualmente; existen librerías o ejemplos de *syntax highlighting* en Tkinter. Si falta tiempo, se puede omitir o solo colorear un par de palabras clave con búsquedas simples.

Recuerda que en la guía se enfatiza que un IDE básico no busca ser un VSCode, sino brindar una interfaz mínima para interactuar con el compilador [88]. Incluso solo con mostrar los errores en una caja de texto ya logras gran parte del objetivo. Si optas por PyQt, podrías usar `QPlainTextEdit` para el código y conectarle una funcionalidad de resaltado personalizando un `QSyntaxHighlighter`. PyQt también facilitaría dibujar un árbol en otro widget (por ejemplo usando `QTreeView` if you convert the parse tree to a model). Pero todo esto es "extra"; cumple primero la funcionalidad esencial:

- **Escribir código .cps** en el editor.
- **Compilar al pulsar botón** (ejecutar análisis) y **resaltar errores con línea** (y ojalá descripción cerca).
- **Ver el árbol sintáctico** de algún modo (texto o gráfico).

Si implementas el IDE, pruébalo con códigos de ejemplo: por ejemplo, un programa correcto debe indicar "Programa sin errores" y quizás mostrar su parse tree; un programa con errores debe resaltar cada error. Esto ayuda a los usuarios (estudiantes) a entender qué está mal y dónde, en tiempo real.

## 10. *Checklist* Técnico del Proyecto

Hemos cubierto todos los componentes necesarios para la fase de Análisis Semántico. Usa la siguiente lista como verificación final de tu proyecto:

- **Gramática ANTLR (Lexer/Parser):** La gramática `Compiscript.g4` ha sido compilada con ANTLR4 y se confirmó que reconoce correctamente construcciones del lenguaje (probada con ejemplos simples) [6] . Los archivos generados (lexer, parser, visitor) están integrados al proyecto.
- **Patrón Visitor implementado:** Se creó una clase `SemanticAnalyzer` (o similar) que extiende el visitor de ANTLR. En ella, se sobreescribieron métodos para las reglas importantes de la gramática, abarcando expresiones, sentencias, declaraciones, etc. Cada método implementa las validaciones semánticas según las reglas definidas (tipos, ámbitos, etc.) y acumula errores cuando corresponde.
- **Tabla de Símbolos funcionando:** Hay una estructura de tabla de símbolos que soporta al menos las operaciones de insertar símbolos y resolver búsquedas, manteniendo la jerarquía de scopes [61] . Está integrada con el visitor (p.ej., `SemanticAnalyzer` mantiene un `currentScope` y lo actualiza al entrar/salir de ámbitos). Se maneja correctamente la declaración de nuevos ámbitos en funciones, clases y bloques anidados [26] [27] .
- **Reglas Semánticas Cubiertas:** Todos los requerimientos semánticos listados (sistema de tipos, coerciones, constantes, resolución de nombres, parámetros/funciones, flujo de control, clases, listas, etc.) están implementados en el visitor. Cada categoría de errores esperados es detectada. Es útil repasar cada punto de la sección 5 y verificar en el código que exista la comprobación correspondiente (por ejemplo, ver que realmente validas tipos numéricos en operadores aritméticos, etc.) [57] [58] .
- **AST (si se implementó):** Si optaste por construir un AST, verifica que esté completo para representar cualquier constructo (nodos para expresiones binarias, unarias, llamadas, declaraciones, etc.). Asegúrate de que el visitor lo construye correctamente y considera la posibilidad de realizar las validaciones sobre el AST en lugar del parse tree, si así lo decidiste. (Si no hay AST, marcar este punto como N/A).
- **Pruebas Unitarias exhaustivas:** Existe un conjunto de pruebas que cubre los casos principales de la semántica. Organizadas por tipo de verificación, aseguran que los casos válidos no generen error y los casos inválidos sí lo generen. Todas las pruebas pasan, lo que indica que el compilador cumple con los requerimientos. Incluye tanto pruebas básicas (ej. operaciones aritméticas válidas/invalidas) como algunas combinadas (e.g., usar una variable de tipo incorrecto dentro de una función) para asegurarse de la robustez [89] [72] .
- **IDE básico (si se implementó):** La interfaz permite editar código, compilar y ver errores/árbol. Se probó manualmente con ejemplos de entrada para confirmar que resalta o muestra adecuadamente los errores semánticos y que no se congela en inputs erróneos (p.ej., manejar excepciones de parseo también). Si no se implementó la GUI, este punto puede ignorarse.
- **Documentación del proyecto:** Hay un archivo README (o informe) donde se explica brevemente cómo usar el compilador, cómo ejecutar las pruebas, y se menciona que se cubrieron todos los requerimientos. Incluso se podría incluir una tabla de requerimientos cumplidos y dónde se evidencian (e.g., mencionar qué test o qué parte del código lo cubre) [90] [58] . Esto es útil para los evaluadores y para tu propio control de calidad.

Recorre esta lista antes de dar por finalizada la fase. Marcar cada elemento te da confianza de que no faltó nada crítico.

## 11. Buenas Prácticas de Desarrollo y Colaboración

Para cerrar, considera estas buenas prácticas al desarrollar el proyecto, especialmente si es en equipo:

- **Commits atómicos y descriptivos:** Realiza *commits* frecuentes, cada uno enfocado en una funcionalidad o fix. Evita mezclar muchos cambios dispares en un solo commit. Por ejemplo, después de implementar la comprobación de tipos en expresiones aritméticas, haz commit con mensaje claro como `"Implementa verificación de tipos en operaciones aritméticas"` [91] . Esto facilita revisar el historial y depurar. Otros commits podrían ser `"Añade manejo de ámbitos anidados"`, `"Corrige validación de return en funciones void"`, etc. Asegúrate de configurar tu Git con tu nombre y email reales, especialmente en proyectos grupales, para atribuir correctamente cada cambio [92] .
- **Colaboración por módulos:** Si trabajan en equipo, dividan las tareas de forma que cada miembro tenga responsabilidad primaria sobre uno o varios módulos del compilador [92] . Por ejemplo, uno se encarga de la **tabla de símbolos**, otro del **visitor de expresiones y declaraciones**, otro de la **parte de clases y objetos**, otro de la **interfaz**. Aunque todos pueden revisar el código de todos, tener un "dueño" de cada parte ayuda a mantener coherencia y distribuir la carga. Integren los aportes vía pull requests si usan plataformas como GitHub, para facilitar la revisión cruzada y asegurar calidad en cada merge [93] .
- **Integración continua y testing:** Intenten que siempre todos los tests pasen en la rama principal (main). Si usan ramas de desarrollo, no las mezclen hasta que lo implementado esté probado localmente. Una práctica útil es configurar GitHub Actions u otro CI para correr automáticamente las pruebas unitarias en cada push/pull request. Así, no se introducen regresiones inadvertidas. Además, si cada commit menciona el requerimiento o issue que aborda (ejemplo: "Implementa [Req. 5] Manejo de listas en asignación"), queda rastro claro de que se atendió cada punto [94] .
- **Documentación y reporte:** Mantengan actualizado el README del repositorio con instrucciones claras de uso (cómo compilar, cómo correr el compilador o IDE, cómo ejecutar tests) [95] . Incluyan una sección listando las funcionalidades implementadas y quizá los requerimientos cumplidos (checklist) [58] , de modo que cualquier persona (o tu profesor) pueda verificar fácilmente el alcance. Si es un proyecto para entregar, suele ser necesario un informe; allí pueden incluir la *checklist* de requerimientos marcando cada uno como implementado y referenciando evidencia (por ejemplo, el nombre del test que lo valida, o fragmentos de código) [90] . Esto demuestra profesionalismo y asegura que nada se quede sin evaluar.
- **Uso de .gitignore:** No versionen archivos innecesarios o derivados. Por ejemplo, los archivos generados por ANTLR (CompiscriptLexer.py, etc.) podrían ignorarse y solo mantener la gramática, ya que cualquiera con ANTLR puede regenerarlos. Lo mismo para archivos temporales, binarios, caches de IDE, etc. Un repositorio limpio con solo fuente y docs es más fácil de manejar [96] .
- **Comunicación y revisión:** Reúnanse periódicamente (o mantengan comunicación por chat) para comentar avances o dificultades. Revisar el código entre ustedes (code review) mejora la calidad y difunde el conocimiento de todo el sistema en el equipo.
- **División en fases**: Aunque nuestro enfoque ha sido la fase semántica, recuerden que en construcción de compiladores se suele integrar todo. Si vienen de haber hecho el análisis sintáctico, integren ese trabajo; y de cara a fases futuras (código intermedio, etc.), mantengan el código modular para extenderlo fácilmente.

- **Checklist final:** Antes de la entrega, repasen toda la funcionalidad con casos de prueba manuales también en la interfaz (si la hay) para garantizar que la experiencia de usuario es buena y que los mensajes de error son informativos (los requerimientos pedían resaltado por línea, por ejemplo, ya lo abordamos) [97] . Una compilación semántica exitosa puede simplemente decir "Programa semánticamente correcto" o similar, para feedback del usuario [98] [99] .

Siguiendo estas prácticas, no solo cumplirán con los requisitos técnicos sino que también presentarán un proyecto bien organizado, mantenible y profesional. ¡Mucho ánimo con la implementación restante y las pruebas! Cada miembro del equipo podrá señalar con orgullo su contribución y habrán creado una herramienta educativa útil además de aprobar el curso.

**Referencias:** Para más detalles, consulta la especificación del lenguaje en *Compiscript.md* [100] , el documento de requerimientos de compilación (fase semántica) y la guía paso a paso proporcionada, de donde extraímos varias de las recomendaciones técnicas [101] [8] . Todos estos recursos respaldan las decisiones de diseño tomadas en esta guía. ¡Happy coding! 🖥

---

[1] ANTLR
https://www.antlr.org/

[2] [5] [7] [21] [28] [43] [44] [45] [48] [52] [53] [69] [100] Compiscript.md
file://file-BjyyQtHLiXSXGQuJMDSJhN

[3] [4] [6] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [22] [23] [24] [25] [26] [27] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [46] [47] [49] [50] [51] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98] [99] [101] Guía para la Fase de Compilación_ Análisis Semántico de Compiscript.pdf
file://file-Nd2dmbTHcJhqoSusnkSN3V