



# Guía para la Fase de Compilación: Análisis Semántico de Compiscript

Esta guía presenta un recorrido detallado, paso a paso, para completar el proyecto de la fase de compilación enfocado en el **Análisis Semántico** del lenguaje **Compiscript**. Se basa en los documentos proporcionados (requerimientos de compilación, especificación del lenguaje Compiscript y su gramática ANTLR), así como en el contenido de referencia del curso CC3071. Está redactada de forma técnica pero accesible, pensada como un manual instructivo para estudiantes universitarios. Incluye recomendaciones de herramientas, fragmentos de código de ejemplo, sugerencias de estructura de proyecto y buenas prácticas de desarrollo en equipo.

## 1. Elección del Lenguaje de Programación para la Implementación

**Recomendación:** Utilizar el lenguaje **Java** para desarrollar el compilador de Compiscript, integrando ANTLR en el proceso. A continuación justificamos esta elección en función de la integración con ANTLR, la facilidad de realizar pruebas unitarias y la posible construcción de una interfaz (IDE) sencilla:

- **Integración con ANTLR:** Java es el lenguaje nativo de ANTLR y el mejor soportado. ANTLR fue creado en Java y genera por defecto código Java para el lexer y parser, lo que asegura compatibilidad inmediata. Aunque ANTLR soporta otros lenguajes (Python, C#, JavaScript, Go, etc.), el *runtime* de Java es el más maduro y con mejor documentación <sup>1</sup>. Esto significa menos inconvenientes técnicos al generar y usar el parser. Además, la comunidad y los ejemplos de ANTLR en Internet suelen estar en Java, lo cual facilita resolver dudas rápidamente. Por ejemplo, el material del curso destaca que Java es “ideal para entornos académicos y desarrollo robusto” <sup>1</sup>, justamente el caso de este proyecto.
- **Facilidad de pruebas unitarias:** Java cuenta con frameworks establecidos como **JUnit** o **TestNG** para escribir pruebas unitarias de manera estructurada. Con JUnit, podemos automatizar la verificación de casos de prueba del compilador (por ejemplo, comprobar que ciertas entradas producen errores semánticos esperados). Otros lenguajes también soportan pruebas (Python con PyTest o unittest, C# con NUnit, etc.), pero Java destaca por la integración sencilla con entornos de desarrollo (IDE como IntelliJ/Eclipse) y la facilidad para ejecutarlas en CI. Adicionalmente, el código generado por ANTLR en Java tiende a ser muy estable, lo que reduce fallos aleatorios y facilita pruebas consistentes.
- **Construcción de una IDE sencilla:** Java ofrece bibliotecas gráficas como **JavaFX** o **Swing** que permiten crear interfaces de usuario de forma razonablemente simple. Podemos desarrollar una aplicación de escritorio básica donde el estudiante pueda escribir código Compiscript, pulsar un botón de “Compilar” y ver resultados (errores resaltados, árbol sintáctico, etc.). Existen componentes ya hechos, por ejemplo en JavaFX, para áreas de texto con resultado de sintaxis o subrayado de errores. En contraste, si eligiéramos Python tendríamos que apoyarnos en librerías externas (PyQt, Tkinter) para la GUI e igualmente integrar el parser de ANTLR (posible con el *runtime* de Python, pero menos documentado). Con Java, toda la solución puede mantenerse en el mismo ecosistema: ANTLR

+ lógica + interfaz gráfica. Además, Java permite empaquetar fácilmente la aplicación (por ejemplo, un **.jar** ejecutable).

- **Otras alternativas:** Python es tentador por su sencillez y rapidez de desarrollo (el curso lo menciona como “fácil de probar y visualizar” <sup>1</sup>). De hecho, ANTLR puede generar un parser en Python si se indica **-Dlanguage=Python**. Sin embargo, la creación de un entorno tipo IDE en Python podría requerir más configuración (por ejemplo, usar PyQt5 para GUI, que tiene su curva de aprendizaje). C# podría ser otra opción (integración con Visual Studio para la interfaz), pero implicaría usar ANTLR en modo C# y depender fuertemente de Windows. Dado el contexto académico y multiplataforma, Java resulta ser la opción más equilibrada.

En resumen, **se recomienda Java** por su excelente integración con ANTLR, disponibilidad de herramientas de prueba y facilidades para una interfaz gráfica educativa. Esta elección nos permitirá centrarnos en la semántica y funcionalidades del compilador sin pelear con el soporte de lenguaje.

## 2. Compilación de la Gramática con ANTLR y Generación de Scanner/Parser

El primer paso práctico es tomar la gramática **Compiscript.g4** proporcionada y utilizar ANTLR para generar el *lexer* (analizador léxico) y *parser* (analizador sintáctico) para nuestro lenguaje. Procedamos paso a paso:

**2.1. Preparar el entorno de ANTLR:** Asegúrate de tener ANTLR4 instalado. Si usas Java, puedes descargar el *jar* completo de ANTLR (por ejemplo **antlr-4.13.0-complete.jar**) y también el *runtime* de ANTLR para Java. Alternativamente, configurar un proyecto Maven/Gradle con la dependencia de ANTLR simplifica el proceso. En un entorno local, puedes añadir el jar de ANTLR al *CLASSPATH* o referenciarlo en el comando.

**2.2. Ejecutar ANTLR sobre la gramática:** Navega al directorio donde esté **Compiscript.g4** y ejecuta el generador. El comando general (siendo **antlr-4.jar** el jar de ANTLR) es:

```
# General (Java target by default)
java -jar antlr-4.13.0-complete.jar -Dlanguage=Java -no-listener -visitor
Compiscript.g4
```

- La opción **-Dlanguage=Java** asegura que el código generado sea Java (no es estrictamente necesario si usas Java, pues es el predeterminado).
- La opción **-visitor** indica a ANTLR que genere las clases base para el *Visitor* (es importante porque implementaremos análisis semántico con *visitors* en vez de *listeners*).
- La opción **-no-listener** es opcional pero conveniente si no planeamos usar *listeners*, así evitamos generar código innecesario.

Al ejecutar este comando, ANTLR leerá la gramática y producirá varios archivos **.java** en el mismo directorio (a menos que especifiques un *output directory* con **-o**). Típicamente se generan: - **CompiscriptLexer.java**: el analizador léxico, encargado de convertir la secuencia de caracteres de entrada en tokens. - **CompiscriptParser.java**: el analizador sintáctico, con métodos que corresponden a las reglas de gramática. - **CompiscriptParserBaseVisitor.java** y **CompiscriptParserVisitor.java**: la clase base con

implementación vacía del patrón Visitor, y la interfaz del Visitor. Estas nos servirán para definir nuestras acciones semánticas. - (Opcionalmente, si hubiera *listeners*, se generarían clases *BaseListener*, *Listener*, pero las omitimos con `-no-listener` ).

**2.3. Revisar la generación:** Una vez generados los archivos, compílalos junto con el *runtime* de ANTLR. Por ejemplo, si trabajas manualmente:

```
# Compilar todos los archivos .java junto con la dependencia de ANTLR
javac -cp .:antlr-4.13.0-complete.jar *.java
```

(En Windows usar `;` en lugar de `:` para separar rutas). Si usas un proyecto Maven, los plugins de ANTLR pueden encargarse de la generación automáticamente durante la fase de generación de fuentes.

**2.4. Confirmar el parser y lexer:** Puedes hacer una prueba rápida de que el parser funciona correctamente. Por ejemplo, escribir un programa sencillo en Compiscript (que esté gramaticalmente correcto) e intentar parsearlo:

```
String inputCode = "...."; // código Compiscript de ejemplo
CharStream input = CharStreams.fromString(inputCode);
CompiscriptLexer lexer = new CompiscriptLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
CompiscriptParser parser = new CompiscriptParser(tokens);
ParseTree tree = parser.program(); // asumiendo que 'program' es la regla
                                   // inicial
System.out.println(tree.toStringTree(parser));
```

Si la gramática y la generación son correctas, `toStringTree` debe imprimir la estructura jerárquica del árbol sintáctico de tu programa de ejemplo (en notación con paréntesis anidados). En este punto ya tenemos las bases: nuestro *scanner* y *parser* funcionan, reconociendo el lenguaje según la gramática. ANTLR nos ha automatizado esta tarea de forma muy eficiente, permitiéndonos concentrar los esfuerzos en las siguientes fases [2](#) [3](#).

*Nota:* Si el parser detecta algún error sintáctico en tu ejemplo, ANTLR invocará sus *manejadores de error* internos y normalmente reportará el error. Esto es normal; puedes personalizar estos mensajes con *ErrorListeners* más adelante si lo deseas, pero inicialmente lo importante es que la gramática esté bien definida.

### 3. Construcción y Visualización del Árbol Sintáctico

Con el parser generado, la siguiente tarea es construir el **árbol sintáctico** (*parse tree*) para el código fuente y visualizarlo, ya que esto ayuda a entender la estructura y depurar la gramática. El árbol sintáctico es una representación jerárquica de la estructura del programa según las reglas gramaticales, donde cada nodo interno corresponde a una construcción del lenguaje y sus hijos son los componentes o subexpresiones que la forman.

**3.1. Generar el árbol sintáctico mediante el parser:** Utilizando el objeto `CompiscriptParser` creado en el paso anterior, al invocar el método de la regla inicial (por ejemplo, `parser.program()`), ANTLR devuelve un objeto `ParseTree` que representa el árbol sintáctico completo. Este árbol contiene todos los nodos etiquetados con sus nombres de regla. Podemos explorar este árbol de varias formas: - Imprimir en texto la representación *LISP* del árbol usando `toStringTree` (ya se mostró un ejemplo). - Recorrerlo manualmente con el patrón Visitor o Listener. - Utilizar métodos proporcionados por ANTLR para iterar hijos, etc.

**3.2. Visualización del árbol:** Para una visualización más intuitiva, ANTLR provee una clase utilitaria `org.antlr.v4.gui.TreeViewer` (cuando se usa Java) que permite mostrar el árbol gráficamente. Por ejemplo, podemos crear un `TreeViewer` pasando la lista de nombres de regla y el parse tree, luego incrustarlo en un `JFrame` de Swing o JavaFX. Otra opción es usar la herramienta *GUI* de prueba de ANTLR (TestRig o la herramienta integrada en plugins de IntelliJ) que grafica el árbol. En un entorno académico, es útil ver el árbol para verificar que la gramática reconoce correctamente cada parte del código.

Supongamos que integras un botón “Ver Árbol” en tu IDE: al presionarlo podrías desplegar una ventana con el árbol. Cada nodo mostrará el nombre de la regla y usualmente el texto del token o símbolo concreto. Esto confirmará visualmente, por ejemplo, que una expresión aritmética se agrupa correctamente respetando precedencias, etc..

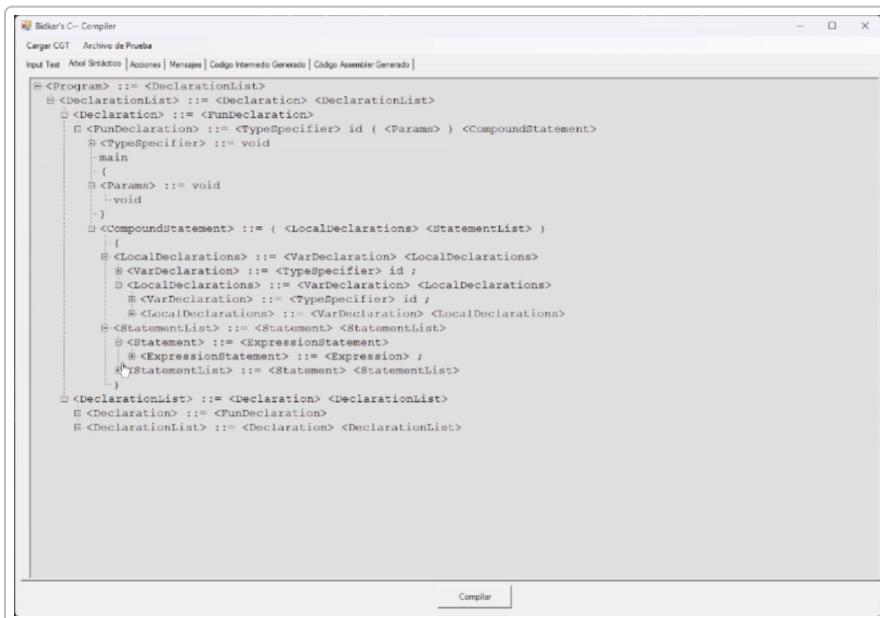


Figura: Ejemplo de visualización de un árbol sintáctico de Compiscript en una interfaz. Cada nodo interno corresponde a una regla gramatical (como `<Declaration>` o `<Statement>`), mostrando cómo se expande según la gramática. Esta representación jerárquica ayuda a validar que la estructura del programa cumple las reglas del lenguaje.

Como se aprecia en la figura anterior, el árbol sintáctico refleja la estructura grammatical del programa fuente: por ejemplo, un nodo `<ExpressionStatement>` puede tener como hijo un nodo `<Expression>` que a su vez tiene nodos operadores e identificadores como hijos. La visualización

confirma que, durante el análisis sintáctico, los tokens han sido organizados correctamente en la jerarquía definida por la gramática.

**3.3. (Opcional) Construir un AST simplificado:** En compiladores a veces se construye un **AST (Abstract Syntax Tree)**, que es una versión simplificada del árbol sintáctico, eliminando nodos innecesarios (p. ej., símbolos de puntuación, o reglas intermedias que no afectan la semántica). Para Compiscript podríamos considerar que el *parse tree* de ANTLR es suficiente, pero si se quiere, se puede crear manualmente un AST en la fase de análisis semántico. Por ejemplo, en cada método visitor podríamos construir objetos nodo propios (clases Java representando nodos de AST) y así tener una representación más manejable para etapas posteriores. Sin embargo, dado el alcance del proyecto, no es obligatorio — trabajar directamente con el parse tree en los visitors es completamente válido.

En resumen, en esta etapa debes lograr que el compilador tome un código de entrada, construya su árbol sintáctico y (al menos para propósitos de desarrollo) seas capaz de visualizarlo o imprimirllo. Esto confirma que el análisis sintáctico funciona correctamente antes de agregar la lógica semántica.

## 4. Implementación del Sistema de Tipos y Reglas Semánticas

Llegamos al corazón del análisis semántico: la **verificación de tipos y otras reglas semánticas** del lenguaje Compiscript. El *sistema de tipos* define qué tipos de datos existen en el lenguaje y qué reglas rigen las operaciones entre ellos. Según los requerimientos, Compiscript incluye tipos básicos (por ejemplo, enteros, flotantes, booleanos, caracteres), probablemente tipos estructurados como *listas*, y entidades como funciones y clases. Cada categoría de constructo tiene reglas semánticas asociadas que debemos implementar:

Las principales funciones del análisis semántico son verificar la **consistencia de tipos** en expresiones y operaciones, y manejar la **tabla de símbolos** para asegurar un uso correcto de identificadores. Recordemos que el análisis semántico detecta errores que el análisis sintáctico no puede, garantizando que las operaciones tengan sentido lógico. A continuación, enumeraremos las reglas semánticas por categoría, y describimos cómo implementarlas (daremos ejemplos de código usando la clase Visitor que desarrollaremos en la siguiente sección):

**4.1. Operaciones aritméticas:** Las expresiones aritméticas (suma, resta, multiplicación, división, módulo, etc.) deben operar sobre tipos numéricos. Por ejemplo, en Compiscript asumiremos que `int` (entero) y `float` (real de punto flotante) son tipos numéricos. Las reglas típicas: - Ambos operandos de una operación aritmética binaria deben ser numéricos. Si uno es entero y otro flotante, es posible que se realice una promoción de tipo (por ejemplo, `int` se convierte implícitamente a `float`) o simplemente se considere `float` como resultado. Depende de la especificación; si no se menciona promoción, podríamos requerir tipos exactamente iguales. - No se pueden aplicar operadores aritméticos a operandos booleanos, cadenas, listas u otros tipos no numéricos. - El resultado de una operación aritmética tendrá un tipo acorde a los operandos: comúnmente,  $\text{int} \otimes \text{int} = \text{int}$ ;  $\text{int} \otimes \text{float} = \text{float}$ ;  $\text{float} \otimes \text{float} = \text{float}$ . ( $\otimes$  representando cualquier operador aritmético). - Operación unaria (ej. negación numérica `-x`): también requiere que `x` sea numérico.

*Implementación:* En el visitor, al procesar un nodo de expresión aritmética (imaginemos un contexto `AddExprContext` para suma/resta), obtendremos los tipos de sus subexpresiones y luego aplicaremos las reglas:

```

@Override
public Type visitAddExpr(CompiscriptParser.AddExprContext ctx) {
    Type leftType = visit(ctx.expr(0));      // tipo del operando izquierdo
    Type rightType = visit(ctx.expr(1));     // tipo del operando derecho
    // Verificación de tipos numéricos
    if (!leftType.isNumeric() || !rightType.isNumeric()) {
        error(String.format("Operación aritmética inválida: %s + %s (operadores
no numéricos en línea %d)",
                            leftType, rightType, ctx.start.getLine()));
        return Type.ERROR; // Type.ERROR puede ser un marcador de tipo inválido
    }
    // Si uno es float, promovemos a float
    if (leftType == Type.FLOAT || rightType == Type.FLOAT) {
        return Type.FLOAT;
    } else {
        return Type.INT;
    }
}

```

(Aquí `Type` puede ser un enum o clase que definamos para representar tipos; `error(...)` es un método auxiliar para registrar errores semánticos).

**4.2. Operaciones lógicas:** Incluyen operadores como `&&` (AND), `||` (OR) y posiblemente `!` (NOT). La regla es que sólo se apliquen a expresiones booleanas: - En `expr1 && expr2` y `expr1 || expr2`, tanto `expr1` como `expr2` deben ser de tipo booleano; el resultado es booleano. - En `!expr`, `expr` debe ser booleano, y el resultado es booleano. - No es válido aplicar operadores lógicos a enteros, strings, etc.

*Implementación:* En el visitor, el contexto de, por ejemplo, una expresión lógica binaria podría llamarse `AndExprContext` / `OrExprContext`. Se validaría así:

```

@Override
public Type visitAndExpr(CompiscriptParser.AndExprContext ctx) {
    Type left = visit(ctx.expr(0));
    Type right = visit(ctx.expr(1));
    if (left != Type.BOOL || right != Type.BOOL) {

        error(String.format("Operación lógica inválida en línea %d: AND requiere
operandos booleanos", ctx.start.getLine()));
        return Type.ERROR;
    }
}

```

```

        return Type.BOOL;
    }
}

```

(Asumimos que `Type.BOOL` representa el tipo booleano.)

**4.3. Operaciones de comparación:** Son expresiones relacionales como `<`, `<=`, `>`, `>=`, `==`, `!=`. Las reglas pueden variar: - Para `<`, `<=`, `>`, `>=`: usualmente se aplican solo a operandos numéricos (int o float). Podría también permitirse en char (comparar código ASCII) si el lenguaje lo definiera, pero en general nos limitamos a numéricos. El resultado de la comparación es booleano. - Para `==` y `!=`: a veces se permiten entre cualquier par de operandos de **tipo compatible** (por ejemplo, puedes comparar dos enteros, o dos booleanos, etc., pero no un entero con un booleano directamente). La definición de compatibilidad de tipos es importante: puede requerir tipos idénticos, o en lenguajes más permisivos, permitir comparaciones cruzadas con conversión implícita (no suele ser el caso en un lenguaje tipado estáticamente simple). - Comparar referencias (por ejemplo dos listas, o objetos) quizás no esté contemplado en Compiscript a menos que definan igualdad referencial, pero podemos asumir que solo tipos primitivos se comparan por valor.

*Implementación:* Por ejemplo, para `expr1 < expr2`:

```

@Override
public Type visitRelationalExpr(CompiscriptParser.RelationalExprContext ctx) {
    Type left = visit(ctx.expr(0));
    Type right = visit(ctx.expr(1));
    String op = ctx.op.getText(); // op podría ser el token <, >, etc.
    if (!left.isNumeric() || !right.isNumeric()) {
        error(String.format("Operador '%s' aplicado a operandos no numéricos
(línea %d)", op, ctx.start.getLine()));
        return Type.ERROR;
    }
    // Podríamos también chequear si left y right son compatibles numéricamente
    // (por ejemplo, permitir int vs float).
    return Type.BOOL; // resultado de cualquier comparación es booleano
}

```

Y para igualdad:

```

@Override
public Type visitEqualityExpr(CompiscriptParser.EqualityExprContext ctx) {
    Type left = visit(ctx.expr(0));
    Type right = visit(ctx.expr(1));
    String op = ctx.op.getText(); // == o !=
    // Regla: tipos deben coincidir exactamente (simplificación)
    if (left != right) {
        error(String.format("No se puede comparar con '%s' tipos distintos (%s
vs %s) en línea %d",

```

```

        op, left, right, ctx.start.getLine()));
    return Type.ERROR;
}
return Type.BOOL;
}

```

(Adaptar los nombres de contexto y detalles según los nombres reales en la gramática Compiscript.)

**4.4. Asignaciones:** La asignación (`=`) es semánticamente válida solo si el **tipo de la expresión del lado derecho es compatible con el tipo de la variable del lado izquierdo**. Esto implica: - La variable debe haber sido declarada antes (verificaremos eso vía la tabla de símbolos en la sección de ámbitos). - Si la variable `x` es tipo T, y se le asigna una expresión de tipo U, entonces U debe ser *asignable* a T. En un lenguaje estático simple, típicamente esto significa que o bien U == T, o existe una conversión implícita de U a T. Por ejemplo, si T es float y U es int, es assignable (promoción de int a float); pero si T es int y U es float, **no** es assignable sin pérdida de información (provocaría error, a menos que el lenguaje permitiera conversión implícita, pero el ejemplo dado sugiere que no se permite). - Asignaciones a constantes (si el lenguaje tiene constantes) no se permiten después de su inicialización.

El material del curso dio un ejemplo ilustrativo: si `a` es int y `b` es float, la asignación `a = b;` debe producir un error de tipos, porque no podemos almacenar un flotante en un entero sin conversión explícita. En cambio, `b = a;` (float = int) podría permitirse con promoción automática.

*Implementación:* En el visitor, al manejar la regla de asignación:

```

@Override
public Type visitAssignStatement(CompiscriptParser.AssignStatementContext ctx) {
    String varName = ctx.ID().getText();
    Type varType = currentScope.lookup(varName); // buscar en tabla de símbolos
    Type exprType = visit(ctx.expr());
    if (varType == null) {
        error(String.format("Variable no declarada: %s (línea %d)", varName,
ctx.start.getLine()));
        return Type.ERROR;
    }
    // Verificar compatibilidad de tipos
    if (!isAssignable(varType, exprType)) {
        error(String.format("Tipos incompatibles en asignación: no se puede
asignar %s a %s (línea %d)",
exprType, varType, ctx.start.getLine()));
        return Type.ERROR;
    }
    return varType; // resultado de la asignación podemos considerarlo el tipo
de la variable
}

```

Aquí `isAssignable(T, U)` implementa la lógica mencionada (por ejemplo, retorna true si `T == U`, o si `T` es `float` y `U` es `int`). Además, vemos la importancia de la tabla de símbolos (`currentScope.lookup`) para obtener el tipo de la variable `varName` declarada. Si devuelve null, significa que la variable no fue declarada en ningún ámbito visible, lo cual es un error semántico que también reportamos.

**4.5. Constantes:** Si Compiscript soporta constantes (por ejemplo palabra clave `const`), las reglas típicas son: - Una constante debe ser inicializada al momento de su declaración, y su valor no puede cambiar posteriormente. - Cualquier intento de asignar a una constante después de declarada es un error. - La expresión inicial asignada a la constante debe ser de tipo compatible con el tipo declarado de la constante (misma regla de asignación).

*Implementación:* Esto implica al declarar la constante, marcar en la tabla de símbolos que es constante. Luego, en el visitor de asignaciones, agregar una verificación: si la entrada en la tabla de símbolos de la izquierda está marcada como constante y la estamos en un contexto de re-asignación, lanzar error “*no se puede modificar una constante*”. Además, en la declaración de constante, verificar la presencia de inicializador.

Ejemplo simplificado:

```
@Override
public Type visitConstDeclaration(CompiscriptParser.ConstDeclarationContext
ctx) {
    String name = ctx.ID().getText();
    Type declaredType = typeFromTypeSpecifier(ctx.typeSpec());
    Type initType = visit(ctx.expr()); // la expresión de inicialización
    if (declaredType != initType) {

        error(String.format("Inicialización de constante %s inválida: se declaró %s pero
se asignó %s (línea %d)",
                           name, declaredType, initType,
                           ctx.start.getLine()));
    }
    // Insertar en tabla de símbolos como constante
    currentScope.insert(name, declaredType, /*isConst= */true);
    return null;
}
```

Y en asignación:

```
if (symbol.isConst) {
    error(String.format("No se puede asignar a la constante %s (línea %d)",
                       varName, line));
}
```

**4.6. Listas (estructuras tipo array/list):** Compiscript soporta listas, que imaginamos funcionan como arreglos o listas de tamaño dinámico. Las reglas semánticas para listas podrían incluir:

- **Literal de lista:** Si el lenguaje permite definir una lista literal, por ejemplo `[1,2,3]` o `["hola","adios"]`, todos los elementos deben ser del mismo tipo (una lista es homogénea). Debes inferir el tipo de la lista a partir de sus elementos. Si mezclan tipos, es error o se debe convertir implícitamente (probablemente error en un lenguaje estático simple).
- **Índice de lista:** Acceder a una lista con un índice (`lista[expr]`) requiere que el índice sea de tipo entero (int). Además, se debe verificar que la variable usada como lista realmente sea de tipo lista.
- **Asignación a posición de lista:** Por ejemplo, `lista[i] = expr;` - aquí hay que checar:
  - `lista` sea de tipo lista de X.
  - `i` sea int.
  - `expr` sea de tipo X (el tipo de elemento de la lista) para poder insertarse en esa posición.
- **Operaciones de lista:** Si hay funciones como `append(element)` o similares, verificar tipos también.

*Implementación:* Podemos representar el tipo de una lista como un tipo parametrizado, por ejemplo `Type.LIST(Type.INT)` o algo similar. En la tabla de símbolos, una variable lista tendría esa información (tipo "lista de enteros", etc.). Entonces:

- Al visitar un literal de lista, iterar por cada expresión elemento y asegurarse de que todos producen el mismo tipo. Si es así, el literal tiene tipo `List<tipoElementos>`. Si está vacío, a veces se considera tipo `List<?>` indeterminado o se requiere un cast, pero probablemente no se manejan listas vacías sin contexto en nuestro caso.
- Al visitar un acceso a lista `lista[expr]`, usar la tabla de símbolos para obtener el tipo de `lista`. Debe ser `List<T>`.
- Visitar `expr` (el índice) y comprobar que su tipo es int; si no, error "El índice de una lista debe ser int".
- El resultado de `lista[expr]` como expresión es de tipo `T` (el tipo de elemento).
- Asignación a lista (si hay una regla para eso), similar a asignación normal pero con este contexto de lista:

```
Type listType = currentScope.lookup(listaNombre); // supongamos listType es
List<T>
if (listType == null or not a list) { ... error ... }
Type indexType = visit(ctx.indexExpr());
if (indexType != Type.INT) {
    error("El índice para acceso a lista debe ser entero...");
}
Type valueType = visit(ctx.valueExpr());
Type elementType = listType.getElementType();
if (!isAssignable(elementType, valueType)) {
    error("Tipo incompatible: no se puede asignar un elemento de tipo
"+valueType+" en una lista de "+elementType);
}
```

- Llamadas a métodos de lista (si existen, e.g. `lista.add(expr)`): verificar que `expr` coincide con el tipo de elemento de la lista.

**4.7. Funciones:** Las funciones introducen varias reglas semánticas:

- **Declaración de función:** Debe especificar un tipo de retorno (o void) y una lista de parámetros con tipos. Al declarar una función, hay que registrar en la tabla de símbolos un símbolo de función que incluya su firma (tipo de retorno y tipos de parámetros) para futuras referencias.
- **Definiciones duplicadas:** Dos funciones no pueden tener el mismo nombre y misma firma en el mismo ámbito (cuidado con sobrecarga, probablemente no aplica en este lenguaje; asumir nombres únicos).
- **Retorno:** Dentro del cuerpo de la función, verificar que todas las

*sentencias return* devuelven un valor del tipo correcto:

- En funciones no-void, **cada** `return expresión;` debe tener un tipo de expresión compatible con el tipo de retorno declarado. Si se encuentra un `return;` vacío en una función con tipo, error.
- En funciones void, cualquier `return` debe ser sin expresión (solo `return;`), y si se encuentra `return expr;` con expr no vacío, es error.
- Posiblemente verificar que una función no-void realmente retorna algo en todas las rutas (esto es más complejo de asegurar estáticamente sin CFG, pero se puede hacer una simple comprobación: por ejemplo, si la última statement del bloque no es un return, advertir).
- **Llamada a función:** Cada invocación `f(expr1, expr2, ...)` debe checar:
  - La función `f` exista (debe estar en la tabla de símbolos).
  - El número de argumentos coincida con el número de parámetros definidos.
  - El tipo de cada argumento sea compatible con el tipo del parámetro correspondiente (por ejemplo, si parámetro es int y se pasa float, ver si hay conversión implícita; generalmente no, debería ser exactamente igual o convertible sin pérdida).
  - El resultado de la llamada en una expresión es del tipo de retorno de la función (sirve para validaciones encadenadas, e.g. usar una llamada en una asignación, etc.).

*Implementación:*

- Al visitar una declaración de función (rule `functionDecl`): - Insertar la función en la tabla de símbolos actual (que podría ser el ámbito global) con su firma. Por ejemplo:

```
String fname = ctx.ID().getText();
Type returnType = typeFromSpec(ctx.returnType());
List<Type> paramTypes = ...; // recorrer la lista de parámetros en ctx.params
if (!globalScope.insertFunction(fname, returnType, paramTypes)) {
    error("Función "+fname+" ya fue definida");
}
```

- Crear un nuevo ámbito (scope) para el cuerpo de la función (ver sección de ámbitos), e insertar en ese ámbito cada parámetro como variable con su tipo.
- Luego visitar el cuerpo (bloque) de la función. Durante esa visita, llevar información del tipo de retorno esperado para validar returns.
- Al visitar una sentencia `return` dentro de una función:

```
Type exprType = ctx.expr() != null ? visit(ctx.expr()) : Type.VOID;
if (currentFunctionReturnType == Type.VOID && exprType != Type.VOID) {
    error("Return con valor en función void (línea "+line+ ")");
}
if (currentFunctionReturnType != Type.VOID) {
    if (exprType == Type.VOID) {
        error("Return vacío en función
"+currentFunctionName+" que debe retornar "+currentFunctionReturnType);
    } else if (!Assignable(currentFunctionReturnType, exprType)) {
        error("Tipo de retorno incorrecto en función
"+currentFunctionName+": se espera
"+currentFunctionReturnType+" pero se retornó "+exprType+" (línea "+line+ ")");
    }
}
```

Aquí se asume que tenemos variables miembro en el visitor, por ejemplo `currentFunctionReturnType` y `currentFunctionName`, que son establecidas al entrar a la función. - Al visitar una llamada de función (ej. rule `callExpr`):

```
String fname = ctx.ID().getText();
FunctionSymbol func = currentScope.lookupFunction(fname);
if (func == null) {
    error("Llamada a función no declarada: "+fname);
    return Type.ERROR;
}
List<ExprContext> args = ctx.exprList().expr(); // por ejemplo
if (args.size() != func.paramTypes.size()) {

error(String.format("Número de argumentos incorrecto al llamar %s: esperados %d,
pasados %d (línea %d)",
fname, func.paramTypes.size(), args.size(),
ctx.start.getLine()));
    // aun así procesamos lo que podamos
}
// Verificar tipos de argumentos
for (int i = 0; i < args.size() && i < func.paramTypes.size(); i++) {
    Type argType = visit(args.get(i));
    Type expected = func.paramTypes.get(i);
    if (!isAssignable(expected, argType)) {

error(String.format("Tipo inválido para el argumento %d en llamada a %s: se
espera %s, se obtuvo %s (línea %d)",
i+1, fname, expected, argType,
ctx.start.getLine()));
    }
}
return func.returnType;
```

De esta forma comprobamos exhaustivamente las llamadas.

**4.8. Clases (y objetos):** Si Compiscript incluye programación orientada a objetos (clases), habrá reglas adicionales:

- **Declaración de clases:** Se debe registrar la clase en la tabla de símbolos (a nivel global). La clase tendrá miembros (atributos y métodos). Es recomendable tener una tabla de símbolos propia para cada clase (representando su *scope* de clase).
- **Atributos:** Verificar que no haya atributos duplicados en la misma clase. Insertar cada atributo con su tipo en la tabla de la clase.
- **Métodos:** Son esencialmente funciones dentro de la clase; las mismas reglas de funciones aplican (tipo de retorno, parámetros, cuerpo). Adicionalmente, puede haber un concepto de `this` implícito; si se requiere, se puede insertar en el scope de método una referencia a la instancia.
- **Herencia:** No sabemos si Compiscript soporta herencia. Si sí, entonces habría que verificar compatibilidad de tipos en asignación cuando están involucradas superclases/subclases (tipo nominal vs tipo real del objeto). Suponiendo que no hay herencia para simplificar.
- **Instanciación/constructores:** Si hay sintaxis para instanciar clases (p.ej. `new MiClase()`), verificar que

la clase exista. Si hay constructor con parámetros, validar como llamada de función. - **Acceso a miembros:** Cuando se usa notación `obj.atributo` o `obj.metodo()`, se debe: - Comprobar que `obj` es de una clase que contiene ese miembro. - Si es atributo, devolver su tipo; si es método, tratarlo como función (posible llamada). - También confirmar visibilidad, aunque si no hay modificadores de acceso en el lenguaje, asumimos todos públicos. - **Polimorfismo/override:** Probablemente fuera de alcance si es un lenguaje sencillo.

*Implementación:* Podríamos tener en la tabla de símbolos una clase `ClassSymbol` que contiene un `Map<String, Symbol>` para atributos y otro para métodos (o métodos también como símbolos). Ejemplo: - Al declarar clase:

```
ClassSymbol classSym = new ClassSymbol(className, currentScope);
if (!globalScope.insertClass(classSym)) {
    error("Clase "+className+" ya definida");
}
// Establecer currentScope a classSym.scope (un nuevo scope para la clase)
currentScope = classSym.getScope();
// Procesar atributos y métodos declarados dentro...
```

- Al acceder a miembro:

```
Type objType = visit(ctx.objExpr());
if (!(objType instanceof ClassType)) {
    error("El objeto no es de una clase válida para acceder a miembros (línea X)");
    return Type.ERROR;
}
ClassSymbol classSym = ((ClassType)objType).classSymbol;
String memberName = ctx.member.getText();
Symbol memberSym = classSym.resolve(memberName);
if (memberSym == null) {
    error(String.format("La clase %s no tiene un miembro llamado %s (línea %d)", classSym.name, memberName, line));
    return Type.ERROR;
}
if (memberSym instanceof VarSymbol) {
    return memberSym.type; // tipo del atributo
} else if (memberSym instanceof FunctionSymbol) {
    // Podríamos soportar obtener una referencia al método, pero si es llamada deberá manejarse en callExpr
    return ((FunctionSymbol)memberSym).getReturnType();
}
```

- En asignación `obj.attr = expr;`: similar a variables globales, pero primero resolver el atributo en la clase de `obj` y luego verificar el tipo.

**4.9. Sentencias de control de flujo:** Incluye `if`, `while`, `for` (si existe), `switch` (si existe). Las reglas generales: - La **condición de un if o while** debe ser de tipo booleano. Si se proporciona un int, float u otro, es error. (En algunos lenguajes C-like un int puede actuar como booleano, pero asumimos tipado estricto: sólo booleano es válido para condición). - En bucles `for`, depende de la sintaxis de Compiscript. Podría haber un for tradicional (`for(init; cond; post)`) donde de nuevo la condición debe ser booleana, y `init` y `post` pueden ser asignaciones o declaraciones (eso se validaría con las reglas de asignación ya descritas). O un for tipo Python (for `x` in `list`) donde se requeriría que `list` sea de tipo iterable (lista). - **Sentencia break / continue**: Si el lenguaje las tiene, deben aparecer únicamente dentro de un bucle; fuera de un loop es error semántico ("break fuera de ciclo no válido"). Esto implica llevar un contexto de si estamos dentro de un loop en el visitor (una variable booleana que se activa al entrar a un `visitWhile` y similares). - **Sentencia return**: Ya la cubrimos en funciones. Una nota: si un `return` aparece fuera de cualquier función (por ejemplo en el main global, si la gramática lo permitiera), habría que marcar error porque no tiene sentido. - `switch / case`: Si existe switch-case, verificar que la expresión del switch es de un tipo que pueda compararse con los case (p. ej., todos los case constantes sean del mismo tipo que la expresión, y que no se repitan valores de case). También el manejo de `default`.

*Implementación:* Ejemplo para if:

```
@Override
public Type visitIfStatement(CompiscriptParser.IfStatementContext ctx) {
    Type condType = visit(ctx.condition());
    if (condType != Type.BOOL) {

        error(String.format("La condición del if debe ser booleana, no %s (línea %d)",
        condType, ctx.start.getLine()));
    }
    visit(ctx.thenBlock);
    if (ctx.elseBlock != null) {
        visit(ctx.elseBlock);
    }
    return null;
}
```

Para while:

```
@Override
public Type visitWhileLoop(CompiscriptParser.WhileLoopContext ctx) {
    Type condType = visit(ctx.condition());
    if (condType != Type.BOOL) {
        error(String.format("La condición del while debe ser booleana (línea
%d)", ctx.start.getLine()));
    }
    // Marcar contexto de loop activo
    inLoop++;
    visit(ctx.body());
    inLoop--;
```

```
        return null;
    }
```

Donde `inLoop` es una variable entera del visitor que incrementamos al entrar a un loop y decrementamos al salir (para saber si un break es válido). Entonces en `visitBreakStatement` podríamos hacer:

```
@Override
public Type visitBreakStatement(CompiscriptParser.BreakStatementContext ctx) {
    if (inLoop <= 0) {
        error(String.format("'break' fuera de un loop (línea %d)",
ctx.start.getLine()));
    }
    return null;
}
```

De forma similar para `continue`.

Con esto cubrimos las categorías principales: **operadores aritméticos, lógicos, relacionales, asignaciones, constantes, listas, funciones, clases y control de flujo**. Cada subregla se traduce a código en el Visitor que valida los tipos y condiciones, emitiendo errores semánticos cuando algo no cumple las reglas del lenguaje. En la siguiente sección veremos cómo organizar la estructura de **ámbitos** y la tabla de símbolos, que es fundamental para que muchas de estas verificaciones funcionen (por ejemplo, saber el tipo de una variable o función referenciada).

## 5. Manejo de Ámbitos y Tabla de Símbolos (Jerarquía de Entornos)

La **tabla de símbolos** es una estructura esencial en el análisis semántico. Su objetivo es almacenar información sobre los identificadores declarados en el programa (variables, funciones, parámetros, clases, etc.), permitiendo consultarlos para verificar usos correctos. Además, dado que los lenguajes tienen **ámbitos** (scopes) – por ejemplo, variables locales dentro de funciones, variables globales, atributos dentro de clases, etc. – la tabla de símbolos debe manejar una jerarquía de entornos. Esto asegura que podamos declarar múltiples variables con el mismo nombre en distintos ámbitos sin conflicto, y que al buscar una variable encontremos la definición más cercana en el ámbito adecuado.

**5.1. Estructura de la tabla de símbolos:** Una implementación común es usar una estructura de datos anidada o una pila de tablas: - Podemos tener una clase `SymbolTable` que internamente use, por ejemplo, un `HashMap<String, Symbol>` para mapear nombres a símbolos (información como tipo, categoría, quizás valor si interpretáramos constantes, etc.). - Cada `SymbolTable` puede tener una referencia a una tabla padre (el ámbito contenedor). Así formamos una cadena jerárquica. Por ejemplo, el ámbito global no tiene padre (es raíz); una función tiene como padre el global; un bloque dentro de esa función tiene como padre la tabla de la función, y así sucesivamente. - Otra forma es mantener explícitamente una pila: cuando entras a un nuevo ámbito, empujas una nueva tabla a la pila; al salir, la sacas. Las operaciones clave son **insertar** símbolos en la tabla actual, **buscar** símbolos comenzando por la tabla actual y subiendo a padres si no se encuentra (regla de *shadowing*: una variable local oculta a una global de mismo nombre, etc.), y **eliminar** al descartar un scope 4 5.

En el curso se mencionó que "la tabla de símbolos suele implementarse como una pila de tablas, cada una correspondiente a un ámbito" <sup>5</sup>. Esta es la aproximación que tomaremos.

**5.2. Clases de símbolo:** Podemos definir clases para distintos tipos de símbolos: - `Symbol` base con propiedades generales (nombre, tipo, quizá flag de constante, etc.). - `VarSymbol` (para variables y parámetros) con tipo de dato. - `FunctionSymbol` con tipo de retorno y lista de tipos de parámetros. - `ClassSymbol` con posiblemente un mapa de miembros y métodos. - Podríamos también tener un `Scope` como abstracción, pero usar `SymbolTable` directamente como Scope es suficiente.

### 5.3. Operaciones típicas:

- **Insertar (declare):** agregar una nueva entrada en el scope actual. Si ya existe un símbolo con ese nombre en el scope actual, se debe reportar error de redefinición en el mismo ámbito (por ejemplo, dos variables locales con el mismo nombre en la misma función no se permiten). Pero es válido si existe en un ámbito superior (eso significa *shadowing*, ocultar la global con una local, lo cual suele permitirse). La inserción típicamente ocurre cuando visitas nodos de declaración (var, func, class, param, etc.). - **Buscar (resolve):** dada una referencia (por ejemplo, al visitar un nodo `x` que es un identificador en una expresión), buscar en la tabla actual; si no está, buscar recursivamente en la tabla padre, y así sucesivamente hasta global. Si no se encuentra, error de *variable no definida*. Esta operación la usaremos en el visitor al ver cada uso de variable, llamada de función, etc., para obtener su tipo y verificar. - **Entrar/salir de ámbito:** cuando entras en un nuevo ámbito (inicio de función, inicio de un bloque `{ ... }` si el lenguaje define que crea un nuevo alcance, etc.), se crea una nueva `SymbolTable` con parent = tabla actual, y se hace la tabla actual. Al salir, se regresa al parent (p.ej., haciendo pop de la pila).

**5.4. Creación de entornos en Compiscript:** Identifiquemos qué constructos crean un nuevo ámbito: - *Ámbito global*: contiene variables globales, funciones, clases definidas a nivel global. - *Cuerpo de función*: nuevo ámbito para variables locales de esa función y sus parámetros. - *Cuerpo de clase*: un ámbito para nombres de atributos y métodos (aunque métodos también tendrán sus propios). - *Bloques interiores*: si el lenguaje soporta `{ ... }` anidados dentro de funciones (como en C/Java), esos también crean ámbito para variables definidas ahí dentro (ej: variables definidas dentro de un `if` en C tienen bloque). - Habría que revisar la gramática Compiscript; si sigue un estilo C, un `CompoundStatement` puede crear un ámbito. - Si es estilo Python (identación) podría ser similar: cada bloque indentado crea scope? (En Python real, cada función crea scope, pero un if no; en muchos lenguajes curly braces sí crean scopes). - Asumamos que los bloques entre `{}` crean ámbito local. - *Estructuras de control con variables propias*: Por ejemplo, un `for(int i=0; ... )` en C tiene `i` en un sub-ámbito. Si Compiscript tiene algo así, manejarlo.

**5.5. Implementación en el Visitor:** Podemos dotar a nuestra clase Visitor de un campo que apunte a la tabla de símbolos actual (`currentScope`), y de otro que apunte al global (`globalScope`). Al iniciar la compilación, `currentScope = globalScope`.

Luego: - Al entrar a una función (visitor de la declaración de función): - Crear una nueva `SymbolTable` (hija de la actual) para el cuerpo. - Insertar los parámetros en ese nuevo scope. - Asignar `currentScope` a la nueva. - *Opcional*: marcar en el visitor `currentFunctionReturnType` para validar returns. - Visitar el bloque de la función. - Al salir del bloque, hacer `currentScope = currentScope.parent` (volver al ámbito anterior). - Similarmente, si tenemos una regla para un bloque `{ ... }` general (por ejemplo, `visitBlock`): - Crear nueva tabla hija, asignar `currentScope`, visitar subnodos, luego restaurar. - En una clase: - Insertar la clase en el `currentScope` (que sería global). - Crear scope de clase (hijo del global) para

miembros. - Cambiar currentScope a scope de clase, insertar símbolos de atributos y métodos. Para métodos, podríamos decidir: insertarlos en scope de clase como FunctionSymbols (y quizá crear scopes hijos de clase para cada método al procesarlos). - Restaurar scope global tras terminar la clase.

**5.6. Información almacenada en la tabla de símbolos:** Por cada símbolo, al menos su nombre y tipo. En casos especiales: - Variables: nombre, tipo, flag constante (si es const), quizás valor si hay evaluación constante (no necesario aquí). - Funciones: nombre, tipo retorno, lista de tipos de params. También podemos guardar referencia al nodo parse o un pointer a su cuerpo (para futura generación de código, etc., aunque no es necesario para solo análisis). - Clases: nombre, referencia a tabla de símbolos de sus miembros, quizá info de superclase.

Con esta infraestructura, nuestras verificaciones semánticas pueden hacer correctamente: - Declaración previa de identificadores antes de uso <sup>6</sup> (p. ej., variable usada sin declarar -> no estará en tabla -> error). - Evitar redeclaraciones inválidas (si insert falla porque nombre existe en scope actual -> error). - Asociar cada identificador en expresiones con su tipo proveniente de la tabla <sup>6</sup>. - Gestionar ámbitos anidados en bloques y funciones (al salir de un ámbito se "olvidan" las variables locales, limpiando la tabla, lo cual reflejaría la liberación de esas variables en tiempo de ejecución teórico).

**5.7. Ejemplo ilustrativo:** Supongamos el siguiente código Compiscript:

```
// Ejemplo hipotético
int x;
function foo(int a) {
    float x = 3.5;
    if (true) {
        int y = a + 1;
        x = y;           // aquí x refiere al float local dentro de foo
    }
    y = 2; // error: y fuera de su bloque
    return x;
}
x = foo(5);
```

- Ámbito global: contiene símbolo `x:int`, `foo:function(int)->float`. - Ámbito de foo (local): contiene `a:int`, `x:float` (sombrando al `x` global dentro de esta función). - Ámbito del if dentro de foo: contiene `y:int`. - Al compilar: - `y = 2;` fuera del if => buscar "y": no está en scope local de foo, tampoco en global, error de variable no definida. - Asignación `x = y;` dentro del if: - `x` se resuelve a float (el de scope de foo), - `y` se resuelve a int (scope if), - asignación float = int -> permitida (promoción int a float) o en nuestro caso, supongamos permitida, estaría bien. - Uso de `a` dentro de foo se resuelve correctamente al parámetro. - Uso de global `x` cuando llamamos `foo(5)`: se resuelve al global `x:int` para asignarle el retorno (que es float) - aquí habría otro error: retornar float y asignarlo a int global, incompatibilidad, a menos que permitamos. Este detalle dependería de si se permiten conversiones; si no, marcaríamos error de tipo en `x = foo(5)`.

Este ejemplo muestra la importancia de la jerarquía de scopes para resolver `x` correctamente (dentro de `foo` es el float local, afuera es el global int). Nuestra tabla de símbolos manejada como pila logra esto automáticamente: la búsqueda de `x` encontrará primero el del ámbito más interno.

**5.8. Estructuras de datos sugeridas:** - Clase `SymbolTable` con: - `parent: SymbolTable` (referencia al parent, null si es global). - `symbols: Map<String, Symbol>`. - Métodos: `insert(Symbol sym)` (retorna false si ya existe en `symbols` mismo nombre), `lookupLocal(String name)` (solo en esta tabla), `lookup(String name)` (busca local, si no parent y así sucesivamente). - Clases `Symbol` como mencionado. Un simple enum `Type` para tipos básicos (INT, FLOAT, BOOL, CHAR, etc.) y quizás una clase `TypeList extends Type` para listas paramétricas, `TypeClass` para clases, etc.

Con la tabla de símbolos y scopes implementados, estaremos equipados para realizar el recorrido del árbol con el visitor verificando todas las reglas semánticas correctamente.

## 6. Análisis Semántico con ANTLR *Visitors*

ANTLR ofrece dos formas principales de recorrer el árbol: *Listeners* (basados en eventos enter/exit) y *Visitors* (basados en llamadas explícitas a cada nodo). En este proyecto usaremos **Visitors**, ya que nos dan más control sobre la navegación y retornos de valores, lo cual es útil para calcular tipos de expresiones. Además, la especificación indicó explícitamente usar Visitors en lugar de Listeners.

**6.1. Configuración del Visitor:** Gracias a la generación con la opción `-visitor`, ANTLR ya nos proporcionó una clase base `CompiscriptParserBaseVisitor<T>` donde `T` puede ser un tipo genérico de retorno (podemos pensar en `<T>` como el tipo que retornarán nuestros métodos al visitar nodos; en nuestro caso, será conveniente usar un tipo `Type` para retornar el tipo resultante de las expresiones, o algún contenedor que permita también indicar null cuando no aplica). También hay una interfaz `CompiscriptParserVisitor<T>` que extiende `ParseTreeVisitor<T>`.

Nuestra estrategia: - Crear una clase, por ejemplo `SemanticAnalyzer` o `TypeChecker`, que extienda `CompiscriptParserBaseVisitor<Type>` (si definimos `Type` como clase/enum para los tipos en Compiscript). - Dentro de esta clase, incluir los campos para la tabla de símbolos (`globalScope`, `currentScope`), mecanismos para recolectar errores (una lista `List<String> errors` donde agregamos mensajes, o simplemente imprimirlos – pero es mejor almacenarlos para poder mostrar varios). - Añadir, quizás, campos contextuales como `currentFunctionReturnType`, `inLoop` contadores, etc., mencionados antes.

**6.2. Override de métodos visitor por regla:** Por cada regla relevante de la gramática, override del método `visitNombreRegla(ctx)`. No es necesario override de *todas* las reglas; por ejemplo, reglas muy básicas (como una regla que simplemente pasa otra, o un terminal) pueden dejarse con la implementación por defecto (que suele ser `return visitChildren(ctx)`). Nos enfocaremos en override de: - Reglas de declaración (variables, constantes, funciones, parámetros, clases). - Reglas de expresiones (aritméticas, lógicas, etc., para hacer cálculo de tipos). - Reglas de sentencias (asignación, return, if, while, etc., para verificaciones contextuales). - Cualquier regla que marque entrada/salida de scope (bloques, funciones, clases) para gestionar `currentScope`.

### 6.3. Ejemplo de definición de la clase visitor (simplificado):

```

public class SemanticAnalyzer extends CompiscriptParserBaseVisitor<Type> {

    // Campos
    private SymbolTable currentScope;
    private final SymbolTable globalScope;
    private final List<String> errors = new ArrayList<>();
    private Type currentFunctionReturnType = null;
    private int inLoop = 0;

    public SemanticAnalyzer() {
        this.globalScope = new SymbolTable(null); // sin padre
        this.currentScope = globalScope;
        // (Opcional: pre-cargar símbolos built-in si hubiera, e.g., funciones
        // estándar)
    }

    public List<String> getErrors() {
        return errors;
    }

    private void error(String msg) {
        errors.add(msg);
        System.err.println(msg);
    }

    @Override
    public Type visitProgram(CompiscriptParser.ProgramContext ctx) {
        // regla inicial - podríamos iterar sobre declaraciones globales
        return visitChildren(ctx);
    }

    @Override
    public Type visitVarDeclaration(CompiscriptParser.VarDeclarationContext
ctx) {
        String name = ctx.ID().getText();
        Type type = typeFromTypeSpec(ctx.typeSpec());
        // Insertar en tabla actual
        if (!currentScope.insert(new VarSymbol(name, type))) {
            error(String.format("El identificador %s ya está declarado en este
ámbito (línea %d)", name, ctx.start.getLine()));
        }
        // Si hay inicializador, validar tipo
        if (ctx.expr() != null) {
            Type exprType = visit(ctx.expr());
            if (!isAssignable(type, exprType)) {
                error(String.format("Asignación incompatible al declarar %s: %s
no se puede convertir a %s (línea %d)",
name, exprType, type,

```

```

        ctx.start.getLine()));
    }
}
return null;
}

@Override
public Type
visitFunctionDeclaration(CompiscriptParser.FunctionDeclarationContext ctx) {
    String fname = ctx.ID().getText();
    Type returnType = typeFromTypeSpec(ctx.returnType());
    // Preparar símbolo de función
    List<Type> paramTypes = new ArrayList<>();
    for (ParamContext p : ctx.params().param()) {
        paramTypes.add(typeFromTypeSpec(p.typeSpec()));
    }
    FunctionSymbol fSym = new FunctionSymbol(fname, returnType, paramTypes);
    if (!currentScope.insert(fSym)) {
        error(String.format("La función %s ya fue declarada (línea %d)",
fname, ctx.start.getLine()));
    }
    // Nuevo scope para el cuerpo de la función
    SymbolTable savedScope = currentScope;
    currentScope = new SymbolTable(savedScope);
    // Insertar parámetros en el nuevo scope
    int i = 0;
    for (ParamContext p : ctx.params().param()) {
        String paramName = p.ID().getText();
        Type paramType = paramTypes.get(i++);
        if (!currentScope.insert(new VarSymbol(paramName, paramType))) {

            error(String.format("Parámetro %s duplicado en función %s (línea %d)",
paramName, fname, ctx.start.getLine()));
        }
    }
    // Configurar contexto de función
    Type savedReturnType = currentFunctionReturnType;
    currentFunctionReturnType = returnType;
    // Visitar el cuerpo (block)
    visit(ctx.block());
    // Restaurar contexto
    currentFunctionReturnType = savedReturnType;
    currentScope = savedScope;
    return null;
}

@Override
public Type visitReturnStmt(CompiscriptParser.ReturnStmtContext ctx) {

```

```

        Type exprType = (ctx.expr() != null) ? visit(ctx.expr()) : Type.VOID;
        if (currentFunctionReturnType == null) {
            error(String.format("'return' fuera de una función (línea %d)",
ctx.start.getLine()));
        } else if (currentFunctionReturnType == Type.VOID && exprType != Type.VOID) {
            error(String.format("La función es void y no debe retornar un valor
(línea %d)", ctx.start.getLine()));
        } else if (currentFunctionReturnType != Type.VOID && exprType ==
Type.VOID) {

            error(String.format("Se esperaba retornar un valor de tipo %s (línea %d)",
currentFunctionReturnType, ctx.start.getLine()));
        } else if (currentFunctionReturnType != null && !
isAssignable(currentFunctionReturnType, exprType)) {
            error(String.format("Tipo de retorno incompatible: se esperaba %s y
se obtuvo %s (línea %d)",
currentFunctionReturnType, exprType,
ctx.start.getLine()));
        }
        return null;
    }

    // ... demás overrides para expresiones, control flujo, etc.

}

```

Este es un ejemplo parcial pero significativo. Aquí vemos:

- Uso de `currentScope` para manejar variables locales, parámetros.
- Uso de `currentFunctionReturnType` para validar retornos.
- Recolección de errores con `error()` (agrega a lista y también imprime, aunque en un IDE preferiríamos solo guardarlos para mostrarlos en la interfaz).
- Implementación de algunas reglas como variable ya declarada en mismo ámbito, o duplicidad de parámetros, etc., que enriquecen los chequeos.

**6.4. Uso del Visitor en el compilador:** Una vez implementada la clase `SemanticAnalyzer` (visitor), la integración es sencilla. En el flujo principal del compilador, después de obtener el parse tree:

```

SemanticAnalyzer checker = new SemanticAnalyzer();
checker.visit(tree); // recorrerá todo el árbol
List<String> errors = checker.getErrors();
if (errors.isEmpty()) {
    System.out.println("Compilación exitosa: no se encontraron errores
semánticos.");
} else {
    System.out.println("Errores semánticos encontrados:");
}

```

```
    errors.forEach(System.out::println);
}
```

Podemos entonces reportar los errores o, en la interfaz gráfica, resaltarlos. Es importante que el visitor no se detenga en el primer error: idealmente debe continuar chequeando todo lo posible para acumular todos los errores. Nuestro diseño permite eso porque en lugar de lanzar excepciones, simplemente registramos mensajes de error y seguimos (eso sí, hay que tener cuidado con errores cascada: por ejemplo, si uso una variable no declarada, su tipo es desconocido; podríamos retornar un `Type.ERROR` especial y propagarlo para evitar más errores derivados de ese).

**6.5. No usar Listeners:** Vale la pena notar por qué preferimos visitors. Un Listener en ANTLR reaccionaría entrando y saliendo de cada nodo automáticamente. Se puede usar para construir tabla de símbolos en la fase de declaración (enter de una declaración de variable para insertarla, etc.) y para validaciones en exist strategy. Sin embargo, para cálculos de tipo es más sencillo con Visitor porque podemos hacer `Type leftType = visit(ctx.left)` y obtener el tipo como retorno, lo cual en Listener implicaría mantener una pila manual de tipos resultantes. Los Visitors facilitan la *propagación de información* (aquí, tipos y contexto) hacia arriba en el árbol.

En conclusión, el Visitor centraliza la lógica semántica: cada método implementa las reglas para una construcción del lenguaje, consultando la tabla de símbolos y utilizando el sistema de tipos para verificar coherencia. Los ejemplos de código mostrados ilustran cómo definir estas reglas. A medida que construyas el visitor, utiliza los requerimientos de semántica como checklist para asegurarte de no omitir ninguno (tipos en operaciones, ámbitos, retorno, etc.).

## 7. Pruebas Unitarias de las Reglas Semánticas

Para garantizar que el análisis semántico funciona correctamente en **todas las categorías** de casos (válidos e inválidos) es crucial implementar **pruebas unitarias**. Estas pruebas nos ayudarán a automatizar la verificación de cada regla requerida según el documento de requerimientos. La idea es tener ejemplos de código que cubran expresiones bien tipadas y mal tipadas, usos correctos e incorrectos de variables, funciones, listas, etc., y comprobar que el compilador (hasta la fase semántica) los clasifica apropiadamente (sin errores cuando deben pasar, o con los errores esperados cuando deben fallar).

**7.1. Configuración del entorno de pruebas:** Si seguimos con Java, podemos usar JUnit. Podemos organizar un módulo de pruebas con casos específicos. Alternativamente, escribir un pequeño programa main que lea archivos de prueba, pero es más formal y mantenible usar pruebas unitarias. Imagina una clase de test `CompiscriptSemanticTests` con métodos `@Test` para cada categoría.

### 7.2. Estructura de una prueba unitaria típica:

Cada prueba deberá: - Preparar un snippet de código Compiscript de ejemplo (como string o cargar de un archivo de texto). - Pasarlo por las fases: lexing -> parsing -> visiting (semántica). - Recoger la lista de errores producida. - Compararla con lo esperado (o simplemente comprobar que esté vacía si esperamos éxito, o que contenga ciertos mensajes si esperamos errores).

Por ejemplo, para probar una asignación válida:

```

    @Test
    public void testAssignCompatible() {
        String code = "int a; float b; b = a;"; // asignar int a float, permitido
        List<String> errors = compileAndGetErrors(code);
        assertTrue(errors.isEmpty());
    }

```

Y para probar una asignación incompatible:

```

    @Test
    public void testAssignIncompatible() {
        String code = "int a; float b; a =
b;"; // asignar float a int, debería dar error
        List<String> errors = compileAndGetErrors(code);
        assertFalse(errors.isEmpty());
        // Opcional: verificar que el mensaje específico esté presente
        boolean found = errors.stream().anyMatch(e -> e.contains("incompatibles en
asignación"));
        assertTrue("Se esperaba error de asignación incompatible", found);
    }

```

Aquí `compileAndGetErrors` sería un método helper que implementamos para correr el código por nuestro compilador hasta fase semántica:

```

private List<String> compileAndGetErrors(String code) {
    // Lex + parse
    CharStream input = CharStreams.fromString(code);
    CompiscriptLexer lexer = new CompiscriptLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    CompiscriptParser parser = new CompiscriptParser(tokens);
    ParseTree tree = parser.program();
    // Semantic check
    SemanticAnalyzer checker = new SemanticAnalyzer();
    checker.visit(tree);
    return checker.getErrors();
}

```

De este modo, cada prueba es concisa. Podemos crear pruebas similares para todas las reglas:

- **Declaración previa de variable:** usar un código con variable no declarada y ver que da error "Variable no declarada".
- **Variable redeclarada en mismo ámbito:** e.g. `{ int x; int x; }` debe dar error.
- **Ámbito permite shadowing:** e.g. `int x; { int x; }` no debe dar error (distinto ámbito).
- **Operadores aritméticos:** probar `1 + 2` (int+int ok), `1 + true` (int+bool error).
- **Operadores lógicos:** `true && false` ok, `1 && 0` error.

- **Comparaciones:** `3 < 5` ok, `"hola" < "mundo"` (si strings no comparables, error).
- **Asignaciones:** ya vimos, también probar asignar booleano a int, etc.
- **Constantes:** intentar modificar una const y verificar error.
- **Listas:**
  - lista homogénea ok: `int[] lista = [1, 2, 3];` (si esa es la sintaxis, por ejemplo) debería pasar.
  - lista heterogénea: `[1, true]` error.
  - indice correcto: `lista[0] = 5;` ok si lista de int.
  - indice no int: `lista["indice"] = 5;` error.
  - asignación tipo distinto: si `lista` es de int, asignar un float: error.
- **Funciones:**
  - Llamada correcta: definir `function f(int a): int { return a; }` y llamarla `f(5)` -> sin error.
  - Llamada con argumentos de más/menos: error.
  - Llamada con tipo incorrecto: `f(true)` cuando espera int -> error.
  - Return correcto vs incorrecto:
    - en función int: todos los caminos retornan int -> ok.
    - retornar tipo equivocado (ej return float en int) -> error.
    - no retornar nada en función no-void -> error.
    - retornar valor en void -> error.
  - Recursividad básica: llamar f dentro de f con correcto número de args (debe resolver bien la función).
- **Clases:**
  - Acceso a atributos: `obj.attr` donde attr existe -> ok, y uno donde no existe -> error.
  - Invocar método inexistente -> error.
  - Asignación de objeto: si hay herencia, etc., (no lo cubriremos si no hay herencia).
  - Instanciación clase inexistente -> error.
  - Asegurar que variables de instancia no se usen sin `obj.` en contextos necesarios (depende de lenguaje, quizás todos atributos requieren prefijo `this` o `obj` ).
- **Control de flujo:**
  - `if` con condición booleana -> ok; con entera -> error.
  - `while` similar.
  - `break` dentro de loop -> ok; `break` fuera -> error.
  - `continue` dentro vs fuera de loop.
  - Anidar loops y breaks para ver que still ok dentro inner etc.
- **Misc:**
  - Combinaciones: por ejemplo, expresión compleja mezclando varias cosas correctas -> ninguna alerta.
  - Ejemplo final de código válido completo (que cumpla todos requerimientos) -> verificar que no hay errores.

Cada prueba debe ser relativamente pequeña y focalizada para saber qué estamos validando. Los mensajes de error precisos ayudan; por ejemplo, el requerimiento indica "deben cubrir todos los requerimientos del documento", lo que significa que por cada enunciado de requerimiento (posiblemente enumerado en `requerimientos_compilación.md`) haya uno o varios tests asociados.

**7.3. Automatización y criterios de éxito:** Considera integrar las pruebas en un sistema de integración continua (CI) si es un proyecto de grupo en Git, para que cada push corra los tests. El éxito será cuando

todos los tests pasen: indicando que todos los errores se detectan y no se generan falsos positivos en casos válidos.

Al desarrollar, es útil aplicar incrementalmente: primero escribir tests que sabes que deben fallar porque la funcionalidad no está hecha, luego implementar la funcionalidad y verlos pasar (enfoque TDD). Por ejemplo: - Escribir test para "variable no definida da error". - Ejecutar, ver que actualmente tal vez no da error (porque aún no implementaste lookup o la regla). - Implementar la lógica en visitor (lookup en cada uso de variable). - Ejecutar de nuevo, test pasa. - Continuar con el siguiente.

**7.4. Ejemplo de verificación manual:** Supongamos ejecutamos nuestra herramienta con un código que tiene varios errores semánticos, esperamos una salida clara. Por ejemplo:

```
int a;  
a = 5;  
b = 3;           // 'b' no está declarado  
if (a) { ... } // 'a' es int, no booleano para la condición
```

Deberíamos obtener algo como:

```
Variable no declarada: b (línea 3)  
La condición del if debe ser booleana, no int (línea 4)
```

Los tests unitarios pueden validar la presencia de estas cadenas. Asegurarse de incluir en los mensajes la información (línea, tal vez columna) ayuda mucho a identificar rápidamente el error.

En conclusión, las pruebas unitarias servirán como **evidencia objetiva** de que cada requerimiento de la fase semántica está cubierto. Al finalizar el proyecto, podrán incluir estos resultados en el informe o repositorio, demostrando que el compilador maneja correctamente tanto programas correctos como los distintos tipos de errores semánticos.

## 8. Diseño e Implementación de un IDE Básico para Compiscript

Un **IDE (Entorno de Desarrollo Integrado)**, aunque sea básico, puede mejorar mucho la experiencia al probar el compilador. La idea es proporcionar una interfaz gráfica donde se pueda escribir código Compiscript, **ver los errores en línea** a medida que se escribe o al compilar, y ejecutar la compilación desde una interfaz amigable. No se busca construir algo tan complejo como VSCode, sino más bien una **herramienta educativa** mínima con las siguientes características: - Un panel de edición de texto con resaltado de sintaxis (opcional) y capacidad de marcar errores (subrayar en rojo o mostrar íconos en las líneas con error). - Un botón o acción de "Compilar" que ejecute el análisis (léxico, sintáctico, semántico) y presente los resultados (árbol sintáctico, mensajes de error). - Posiblemente pestañas o áreas para distintas vistas: código fuente, árbol sintáctico visual, lista de errores, incluso código intermedio/assembler en fases posteriores (según vimos en los ejemplos visuales proporcionados).

**8.1. Elección de tecnología para la interfaz:** Dado que recomendamos Java, la opción natural es JavaFX para la GUI (o Swing, pero JavaFX es más moderna y tiene mejor soporte para estilizar texto). Con JavaFX se

puede usar el componente `CodeArea` de la biblioteca RichTextFX para manejar fácilmente texto con colores y estilos, aunque para un proyecto académico quizá baste con un `TextArea` y marcar manualmente errores.

Otra alternativa es crear un pequeño cliente web (HTML/CSS/JS) usando, por ejemplo, la librería Monaco Editor (que es la base de VSCode) o Ace Editor para la edición de código, y un backend Java que corra el compilador. Sin embargo, esto añade complejidad de arquitectura (comunicación web, etc.). Así que asumamos una aplicación de escritorio JavaFX monolítica: - Ventana principal con un menú (por ejemplo, opciones de cargar archivo, guardar, etc. si se desea). - Área de texto principal para el código fuente. - Quizá un panel inferior o lateral para mensajes de error. - Posibilidad de pestañas/áreas para árbol sintáctico y resultados.

**8.2. Mostrar errores en línea:** Hay varias maneras: - La más simple: en la propia área de texto, subrayar o resaltar la porción que tiene error. Esto requiere conocer la posición (offset) de los errores en el texto. ANTLR nos da la información de línea y columna de cada token/nodo (por ejemplo, `ctx.start.getLine()` y `ctx.start.getCharPositionInLine()`). Con eso podemos, en la interfaz, resaltar desde ese offset hasta el fin de la palabra/token con un estilo rojo ondulado. - Alternativa: marcar toda la línea en rojo, o poner un ícono de advertencia en el margen. Implementar un *gutter (margen)* con íconos es más complejo; subrayar texto es más directo con estilos. - También se puede simplemente listar los errores en un `TextArea` aparte con el número de línea, pero la frase "ver errores en línea" sugiere que idealmente estén indicados junto al código.

Un término medio: mostrar errores en un panel aparte *pero* al hacer clic en el error, llevar el cursor a la línea correspondiente. Esto es más fácil y aún útil.

Los ejemplos visuales del proyecto anterior (Bikdar's C-- Compiler) muestran un panel de mensajes con líneas y descripciones de error. En la imagen, por ejemplo, aparece "*Variable no definida. Variable: i Línea: 6*" y "*Condición debe ser INT o BOOL. Línea: 7 Columna: 1*". Eso está en la pestaña "Mensajes", mientras el código estaría en "Input Text". Ellos sumaron un conteo "Errores de Comprobación Estática: 3". Podemos inspirarnos en este formato.

### 8.3. Implementación paso a paso:

1. **Configurar el proyecto con JavaFX:** Asegúrate de incluir las librerías JavaFX necesarias. Crea una clase `Main` que extienda `Application` (if using JavaFX) con un método `start(Stage primaryStage)`.

2. **Diseñar la UI:** Por simplicidad, sin complicar con FXML, se puede armar en código:

3. Un `BorderPane` principal.

4. En el centro, un `TabPane` con al menos dos tabs: "Código" y "Árbol Sintáctico". En "Código" habrá un `StackPane` o directamente un `AnchorPane` con el `TextArea` de código. En "Árbol Sintáctico", podríamos colocar un `ScrollPane` donde incrustaremos un nodo de árbol (quizá generado por TreeViewer de ANTLR o manual).

5. Abajo, un área de texto de solo lectura para mensajes de compilación (errores).

6. Un botón "Compilar" (por ejemplo, ubicado en la parte inferior o en la barra de menú).

7. Menú superior (opcional): para abrir/guardar archivos de código, etc.

8. **Resaltado de sintaxis (opcional):** Se puede ignorar para enfocarse en la funcionalidad, o aplicar estilos básicos: e.g., palabras clave con un color, literales con otro. Con RichTextFX CodeArea se podría hacer tokenización coloreada, pero supone esfuerzo extra. No es estrictamente pedido, así que podemos dejar texto negro simple para el editor.

9. **Acción de Compilar:** Cuando el usuario hace clic en "Compilar":

10. Tomar el contenido del TextArea (el código fuente).

11. Correr el compilador: lex+parse+visit (tal como en tests).

12. Capturar errores semánticos. También podríamos interesarnos en errores sintácticos:

- Si el parser encontró errores sintácticos, también hay que mostrarlos. ANTLR por defecto envía errores sintácticos a stderr. Podemos redirigirlos usando un `BaseErrorListener` personalizado que agregue a nuestra lista de errores algo como "Error sintáctico en línea X: ...". Incluir esto haría el IDE más completo (porque de nada sirve un analizador semántico si la sintaxis ni siquiera parsea; hay que informar ambos).

13. Llenar el panel de mensajes con los errores encontrados (o el mensaje "Programa compilado exitosamente" si ninguno).

14. También, si no hubo errores sintácticos, generar y mostrar el árbol sintáctico en la pestaña correspondiente:

- Podríamos usar el `TreeViewer` de ANTLR para obtener un `javax.swing.JPanel` con el dibujo del árbol. Con JavaFX, integrar Swing es posible con `SwingNode`. Otra opción: construir manualmente un árbol JavaFX (clase `TreeItem` y `TreeView`) representando la estructura del parse tree. Puede ser un reto pero factible: recorrer recursivamente el `ParseTree`, crear `TreeItem` por nodo con el texto de la regla.
- O incluso, mostrar la representación textual del árbol en un TextArea monoespaciado en esa pestaña.

15. Resaltar en el editor las líneas con error:

- Podríamos, por cada error con número de línea, marcar esa línea con un fondo coloreado. En un TextArea simple no es trivial colorear *background* por line (tendríamos que insert styled segments).
- Alternativamente, mover el cursor a la primera línea de error automáticamente, para alertar al usuario.
- Con un componente rico, se podría underline, pero si estamos con TextArea normal, podríamos insert a caret and maybe a tooltip... Esto es más complejo sin libs, así que listarlos en panel de mensajes con line numbers es aceptable.

En los ejemplos de Bikdar's, parece que no subrayan en el editor, solo listan en "Mensajes". Eso cumple "ver errores en línea" en el sentido de "ver con número de línea".

*Recomendación:* Implementar la lista de errores con doble clic para saltar a la línea: - Mostrar errores en un `ListView<String>` o `TextArea`. - Si es ListView, cada item es un error. Se puede agregar un handler a selection: on select, parsear la cadena para extraer número de línea, y posicionar el caret del editor en esa

línea (ej: `textArea.positionCaret(offsetDeLinea(linea))`). - Este approach mejora la navegación de errores.

1. **Ejecución periódica o manual:** Podríamos compilar solo al hacer clic, o agregar funcionalidad de compilación en tiempo real (cada vez que se deja de tipear un segundo, compilar automáticamente). Esto último es agradable (como los IDEs reales que subrayan al vuelo), pero para un proyecto académico, un botón manual es suficiente y más fácil (evita compilar en medio de escribir código incompleto).

**8.4. Código de ejemplo para resaltar una línea en JavaFX TextArea:** Aunque es opcional, supongamos queremos colorear la línea con error. Una estrategia: - TextArea en JavaFX no soporta múltiples estilos en texto fácilmente (diferente a TextFlow). - Sin librerías externas, se podría hacer lo hacky: duplicar el texto en un TextFlow con line breaks y each line in a Text node with red background if error. Pero es mucho. - Mejor: simplemente seleccionar la línea entera usando caret positions:

```
int lineWithError = ...;
int start = textArea.position(lineWithError, 0).toOffset(); // need method to
get offset from line
int end = textArea.position(lineWithError, lineTextLength).toOffset();
textArea.selectRange(start, end);
// then maybe apply a style? But TextArea selection has a fixed style (blue
highlight typically).
```

Esto no es ideal.

Dado el tiempo, dejarlo en listar con número de línea es aceptable. Puedes escribir en la guía que "lo ideal sería subrayar, pero en aras de simplicidad, mostraremos los errores con sus líneas en un panel aparte, donde se pueden consultar" – que es de hecho lo que se ve en los ejemplos visuales.

**8.5. Mostrar evidencias gráficas:** Podrías tomar capturas de pantalla de tu IDE funcionando (como hicieron con Bikdar's). En nuestro caso textual, describiremos con detalle.

**Resumen de la interfaz:** - **Ingreso de código:** El usuario escribe o carga un archivo fuente de Compiscript en el área de texto. - **Botón Compilar:** Al presionar, se ejecutan todas las fases hasta semántica. Si no hay errores sintácticos/semánticos, se puede indicar "Programa Aceptado" y quizás habilitar fases siguientes (generación de código intermedio, etc., si existieran). Si hay errores, se listan. - **Mensajes de error:** Incluyen tipo de error y ubicación. Por ejemplo, "Variable no definida: i (Línea 6)", "Tipos en la asignación no concuerdan (Línea 6)", "Condición debe ser INT o BOOL (Línea 7)", etc., con un contador final de errores. - **Visualización del árbol sintáctico:** En otra pestaña o panel, mostrar el árbol para que el estudiante pueda relacionar el código con la estructura parseada.

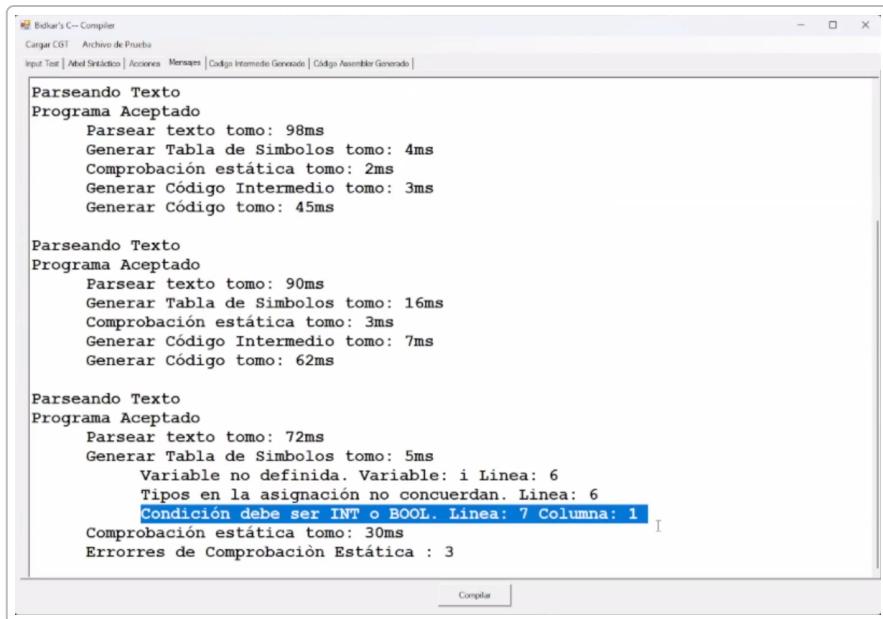


Figura: Ejemplo de salida de la compilación en la interfaz. Se muestran los mensajes de la fase semántica con detalles de línea/columna, incluyendo errores de variable no definida, incompatibilidad de tipos en asignación y tipo de condición incorrecto. En este caso, el compilador detectó 3 errores de comprobación estática (análisis semántico).

La figura anterior ilustra cómo el IDE podría listar los errores tras compilar: en azul está resaltado un error específico "Condición debe ser INT o BOOL. Línea: 7 Columna: 1", que corresponde a usar un tipo incorrecto en la condición de una estructura de control. De esta manera el programador puede identificar rápidamente dónde necesita corregir el código. La interfaz también muestra los tiempos consumidos en cada fase (opcional, pero interesante informativamente) y un conteo total de errores encontrados.

**8.6. Herramientas auxiliares recomendadas:** - *RichTextFX CodeArea*: si se desea mejorar la edición con colores y estilos, esta librería de código abierto facilita aplicar spans de estilo por rangos de texto, ideal para resaltar errores con un subrayado rojo ondulado (usando CSS). - *ANTLR IDE plugins*: Aunque no es exactamente parte del IDE final para el usuario, los desarrolladores pueden usar el plugin de ANTLR en IntelliJ para depurar la gramática y visualizar árboles durante el desarrollo. - *DockFX or ControlsFX*: librerías JavaFX con controles mejorados, en caso de querer hacer el UI más sofisticado (por ejemplo, paneles acopiables). - *JUnit (para integrar pruebas dentro del IDE)*: Podrías crear en la interfaz un botón "Ejecutar tests" que corra los tests unitarios y muestre un resumen, aunque esto es extra; normalmente se ejecutan fuera del app.

**8.7. Conclusión de la parte de IDE:** La implementación de un IDE básico consolidará tu compilador en una herramienta utilizable. Si bien el foco del curso es la construcción del compilador, la interfaz demuestra un entendimiento completo del proceso al brindar una experiencia más real. Además, al usar el compilador vía interfaz, podrían descubrirse casos no considerados (sirve como prueba manual). Empieza con algo mínimo (editor + botón + log de errores) y luego iterativamente mejora si el tiempo lo permite (agregar la visualización de árbol, etc.). En todos los casos, mantén la interfaz intuitiva: por ejemplo, limpiar los errores viejos antes de una nueva compilación, indicar claramente si la compilación fue exitosa, etc.

## 9. Buenas Prácticas de Repositorio: Organización y Colaboración en Equipo

Finalmente, unas recomendaciones sobre cómo mantener el proyecto organizado en el repositorio y evidenciar las contribuciones de cada integrante del grupo:

- **Estructura de carpetas limpia:** Organiza el código fuente, los archivos de gramática, pruebas y recursos de forma estructurada:

```
compiscript-compiler/
├── src/
│   ├── main/
│   │   ├── Compiscript.g4      (gramática ANTLR)
│   │   └── edu/uvg/compiscript/ (paquete de código fuente Java, por
ejemplo)
│   │       ├── Main.java      (clase con método main para interfaz)
│   │       ├── SemanticAnalyzer.java (visitor)
│   │       ├── SymbolTable.java, Symbol.java, ... (estructuras semánticas)
│   │       └── ... otras clases
│   └── test/
│       └── edu/uvg/compiscript/
│           ├── CompiscriptSemanticTests.java (pruebas unitarias)
│           └── ... otros tests
└── lib/                                (jar de ANTLR y quizá JavaFX if needed)
└── README.md                            (documentación del proyecto, instrucciones)
└── examples/                           (opcional, código de ejemplo .cmp para
probar)
└── docs/                               (opcional, documentación adicional, reportes,
etc.)
```

Esta es una sugerencia asumiendo Java y un build tool. Si usan Maven, la estructura `src/main/java` y `src/test/java` funcionará con su pom.xml. Incluye la gramática en el repo para transparencia, así como cualquier script (por ejemplo, `buildLanguage.sh` si tuvieran uno para ANTLR).

- **Control de versiones y commits atómicos:** Cada funcionalidad o corrección debe ir en un commit separado con un mensaje claro. Evitar *commits* grandes mezclando muchas cosas. Por ejemplo: "Implementar verificación de tipos en expresiones aritméticas", "Agregar manejo de ámbitos anidados", "Corregir bug en return statements", etc. Esto facilita el seguimiento del progreso y la revisión.
- **Evidencia de contribuciones por integrante:** En proyectos grupales, los profesores a veces revisan el historial de Git para ver quién hizo qué. Para demostrar contribuciones:

- Cada miembro debe hacer commits con su usuario (configurar correctamente nombre y correo en Git).
- Pueden dividir tareas por archivos o módulos. Ejemplo: alguien se encarga de la tabla de símbolos, otro de las reglas de expresiones y pruebas de esas, otro de la interfaz, etc. Aunque todos colaboren en todo un poco, encontrar un *ownership* primario de ciertas partes es bueno.
- Usar *pull requests* en GitHub (si el repo es público/privado en GitHub) también deja constancia de revisión y de quién escribió código.
- Evitar la situación de un solo miembro haciendo todos los commits. Si por flujo de trabajo uno programa y otro no sabe Git, al menos que se sienten juntos para escribir el código de esa persona bajo su cuenta.
- En el archivo README.md del repo, podrían incluir una sección de "Contribuciones" donde describan brevemente qué realizó cada miembro (esto es explícito y útil para el docente).
- **Documentación en el repositorio:** Mantén actualizado el README con:
  - Descripción del proyecto.
  - Cómo compilar y ejecutar el compilador (instrucciones para lanzar el IDE, o correr por línea de comando).
  - Ejemplos de uso.
  - Estado de implementación (qué features están completas, qué falta si algo).
  - Resultados de pruebas (quizá un cuadro resumido).
  - Cualquier referencia a los documentos base (pueden mencionar que se siguió tal requerimiento y tal).

Esto ayuda a cualquier evaluador a entender rápidamente el alcance y a reproducir resultados.

- **Gestión de ramas (branching):** Opcional pero recomendado en equipo. Podrían tener una rama principal (main/master) donde solo se hace merge de lo que está estable, y ramas de desarrollo por funcionalidad o por persona. Ejemplo: semantic-analysis, parser-implementation, guid-dev etc., que luego integran. Esto permite trabajar en paralelo sin pisarse y luego fusionar con revisiones de código conjuntas.
- **Uso de issues y Kanban:** En GitHub, crear Issues para cada requerimiento/feature y asignarlos a miembros puede ser útil. Además, usar Projects (kanban style) para mover tareas de *To do* a *In progress* a *Done* visualiza el avance. Esto no es obligatorio, pero muestra un alto nivel de organización.
- **Evitar archivos innecesarios:** Añadir un .gitignore adecuado: por ejemplo, ignorar archivos generados por ANTLR (\*.class), o si generan código, aunque el código generado tal vez quieran incluirlo, no es necesario porque se puede generar de nuevo), archivos de entorno (como .idea de IntelliJ, etc.), binarios, etc. El repositorio debe contener principalmente código fuente y documentación, no compilados. La limpieza del repo será valorada.
- **Commits vinculados a requerimientos:** Si el documento de requerimientos tiene numeración, podrían mencionar en los mensajes de commit "#Req5 completado: Manejo de listas" etc., para dejar claro qué requerimiento se aborda.

- **Verificación final en README o informe:** Una buena práctica es tener en la documentación una checklist de requerimientos y marcarlos como implementados. Por ejemplo:

Requerimiento	Implementado	Evidencia (clase/método de código, prueba)
Verificación de tipos aritméticos	Sí	<code>visitAddExpr</code> en <code>SemanticAnalyzer.java</code> , Test <code>testArithmeticTypes</code>
Manejo de variable no declarada	Sí	<code>lookup</code> en <code>SymbolTable</code> y uso en <code>visitIdExpr</code> , Test <code>testUndeclaredVar</code>
...	...	...

Esto demuestra explícitamente que nada se quedó sin atender.

Siguiendo estas prácticas, lograrán un proyecto bien estructurado, fácil de navegar para los evaluadores y mantendrán un registro claro del aporte de cada integrante. Un repositorio limpio y bien documentado refleja profesionalismo y facilita tanto el desarrollo colaborativo como la revisión y mantenimiento futuros.

---

**Conclusión:** Al completar esta fase de análisis semántico, habrán dotado a su compilador de la inteligencia para entender el significado y la coherencia del código, más allá de la sintaxis. Cuentan ahora con: - Un lenguaje claramente definido (Compiscript) con su gramática ANTLR compilada. - Un sistema de tipos implementado que evita operaciones inválidas y usos erróneos de datos. - Manejo correcto de ámbitos y una tabla de símbolos funcional, garantizando la correcta resolución de identificadores [6](#). - Un recorrido del árbol sintáctico con Visitors que efectúa todas las comprobaciones semánticas necesarias. - Conjunto de pruebas que aseguran la calidad de estas funcionalidades. - Y una interfaz de usuario básica que integra todo, haciendo la herramienta accesible y demostrando su funcionamiento de manera visual.

Con esta base sólida, las siguientes fases (como generación de código intermedio o optimizaciones, si forman parte del curso) podrán apoyarse en un front-end robusto. ¡Enhorabuena por llegar hasta aquí, y mucho éxito con las etapas finales del compilador!