

## README Tarea 1: Preparación de la Gramática

### ¿Qué es esta fase?

La preparación de la gramática consiste en definir correctamente las reglas sintácticas del lenguaje Compiscript utilizando el formato ANTLR (`Compiscript.g4`). Se busca que las reglas sean claras, completas y sin ambigüedades.

### ¿Qué hace?

- Revisa la estructura BNF.
- Valida ambigüedades en la gramática.
- Asegura que las reglas sigan una jerarquía adecuada.

### ¿Qué recibe?

- Archivo `Compiscript.g4`

### ¿Qué salida produce?

- Versión limpia y validada de `Compiscript.g4`, lista para compilación con ANTLR.

### ¿De dónde viene la información?

- Diseño del lenguaje Compiscript.
- Reglas definidas por el equipo docente.

### ¿Hacia dónde va la información?

- Al proceso de compilación con ANTLR para generar el parser.

### Ejemplo:

```
expr: expr '+' expr    # Suma
      | expr '-' expr    # Resta
      | INT               # Número entero
      ;
```

Problema: esta definición es **ambigua** (no asocia precedencia).

Solución:

```
expr: term (( '+' | '-' ) term)*;
term: factor (( '*' | '/' ) factor)*;
factor: INT | '(' expr ')';
```

---

## README Tarea 2: Compilación con ANTLR y Generación del Parser

### ¿Qué es esta fase?

Esta etapa toma la gramática validada de Compiscript y la compila usando ANTLR, generando automáticamente los componentes `Lexer` y `Parser`.

### ¿Qué hace?

- Compila la gramática `.g4`.
- Genera los archivos fuente del analizador léxico y sintáctico.

### ¿Qué recibe?

- El archivo `Compiscript.g4` validado.

### ¿Qué salida produce?

- Archivos generados:
  - `CompiscriptLexer.py`
  - `CompiscriptParser.py`
  - `CompiscriptVisitor.py` (u otros según lenguaje)

### ¿De dónde viene la información?

- Archivo de gramática (`.g4`).

### ¿Hacia dónde va la información?

- Al código del compilador para parseo y análisis semántico.

### Ejemplo de compilación en Python:

```
antlr4 -Dlanguage=Python3 Compiscript.g4
```

Archivos generados:

```
CompiscriptLexer.py  
CompiscriptParser.py  
CompiscriptVisitor.py
```

# README Tarea 3: Construcción del Árbol Sintáctico y Visualización

## ¿Qué es esta fase?

Es el proceso donde el compilador convierte el código fuente en una estructura jerárquica (AST) que representa su significado sintáctico.

## ¿Qué hace?

- Usa el parser generado para recorrer el código fuente.
- Construye el AST usando el patrón Visitor de ANTLR.
- Puede generar una visualización del árbol.

## ¿Qué recibe?

- Código fuente `.cps` válido.

## ¿Qué salida produce?

- Objeto de árbol sintáctico.
- (Opcional) Representación visual en formato gráfico.

## ¿De dónde viene la información?

- Lexer y Parser generados por ANTLR.

## ¿Hacia dónde va la información?

- Módulo de análisis semántico.

### Ejemplo:

```
visitor = CompiscriptVisitor()
result = visitor.visit(tree)
```

Visualización posible con Graphviz:

```
import graphviz
# generar nodos del árbol y exportar
```

## README Tarea 4: Implementación del Sistema de Tipos

### ¿Qué es esta fase?

Implementa las reglas que permiten validar que las operaciones se realizan entre tipos de datos compatibles.

### ¿Qué hace?

- Recorre el AST.
- Verifica reglas de tipos en operaciones (aritméticas, lógicas, asignaciones).
- Reporta errores de tipo.

### ¿Qué recibe?

- Árbol sintáctico anotado.

### ¿Qué salida produce?

- Mensajes de error en consola o IDE si hay conflictos de tipo.

### ¿De dónde viene la información?

- Gramática y reglas definidas por el lenguaje.

### ¿Hacia dónde va la información?

- Sistema de errores y validación.

### Ejemplo:

```
let x: integer = true;
```

Debe producir un error: Asignación de boolean a variable de tipo integer.

---

## README Tarea 5: Manejo de Ámbitos y Tabla de Símbolos

### ¿Qué es esta fase?

Crea estructuras que permiten rastrear las declaraciones y usos de identificadores en diferentes contextos del programa.

### ¿Qué hace?

- Implementa entornos jerárquicos.
- Crea la tabla de símbolos.

- Detecta errores como variables no declaradas o duplicadas.

### ¿Qué recibe?

- Información de nodos del AST.

### ¿Qué salida produce?

- Estructura de tabla con validaciones correctas o errores.

### ¿De dónde viene la información?

- Declaraciones, funciones, clases, bloques del código fuente.

### ¿Hacia dónde va la información?

- Al módulo de análisis semántico.

#### Ejemplo:

```
print(nombre);
```

Si `nombre` no fue declarado antes, debe reportar un error.

---



## README Tarea 6: Análisis Semántico Completo usando Visitor

### ¿Qué es esta fase?

Es la combinación de todos los análisis anteriores en una sola pasada o conjunto de pasadas sobre el AST.

### ¿Qué hace?

- Ejecuta validaciones completas del programa.
- Integra verificación de tipos, tabla de símbolos y reglas semánticas.

### ¿Qué recibe?

- Árbol sintáctico generado por el parser.

### ¿Qué salida produce?

- Lista de errores semánticos.

### ¿De dónde viene la información?

- Todos los módulos de compilación anteriores.

## ¿Hacia dónde va la información?

- Consola o interfaz gráfica del IDE.

### Ejemplo:

```
const PI;
```

Error: `const` debe inicializarse al momento de declararse.

## ■ README Tarea 7: Creación de Pruebas Unitarias

### ¿Qué es esta fase?

Incluye el desarrollo de archivos `.cps` con ejemplos válidos e inválidos para validar el comportamiento del compilador.

### ¿Qué hace?

- Automatiza la validación del análisis semántico.
- Permite comparar resultados esperados vs resultados obtenidos.

### ¿Qué recibe?

- Casos de prueba escritos por el equipo o el docente.

### ¿Qué salida produce?

- Reportes de éxito o fallo por cada prueba.

### ¿De dónde viene la información?

- Módulos del compilador.

## ¿Hacia dónde va la información?

- Consola de pruebas o CI.

### Ejemplo:

Archivo `prueba.cps`:

```
function f() {
    return;
```

```
    }  
    return 1;
```

Debe fallar porque el `return` final no está dentro de una función.

---



## README Tarea 8: Desarrollo del IDE Interactivo

### ¿Qué es esta fase?

Consiste en crear una interfaz visual donde el usuario pueda escribir código, compilar y ver errores directamente desde una aplicación.

### ¿Qué hace?

- Permite edición de código.
- Muestra errores en tiempo real o al compilar.

### ¿Qué recibe?

- Código fuente del usuario.

### ¿Qué salida produce?

- AST, errores y resultados visuales.

### ¿De dónde viene la información?

- Entrada del usuario.

### ¿Hacia dónde va la información?

- Al backend de compilación (lexer, parser, visitor).

### Ejemplo:

Editor con botón `Compilar` que resalta errores como:

```
Línea 3: variable "x" no declarada.
```



## README Tarea 9: Documentación Técnica y README(s)

### ¿Qué es esta fase?

Redactar documentación clara, completa y actualizada de cada etapa del compilador.

### **¿Qué hace?**

- Explica el funcionamiento del compilador.
- Detalla entradas, salidas, procesos internos.

### **¿Qué recibe?**

- Código fuente de cada módulo.

### **¿Qué salida produce?**

- Archivos `README.md`, tablas, diagramas.

### **¿De dónde viene la información?**

- Cada miembro del equipo y sus tareas.

### **¿Hacia dónde va la información?**

- Documentación final del repositorio.

### **Ejemplo:**

```
## Árbol Sintáctico
- Se genera a partir del parser.
- Se recorre con Visitor.
```

---

## **README Tarea 10: Administración del Repositorio y Evidencias**

### **¿Qué es esta fase?**

Se encarga de gestionar el control de versiones y registro de contribuciones al proyecto.

### **¿Qué hace?**

- Configura el repositorio.
- Controla ramas y pull requests.
- Supervisa commits por integrante.

### **¿Qué recibe?**

- Código fuente y documentación.

### **¿Qué salida produce?**

- Historial limpio de versiones y evidencias.

## **¿De dónde viene la información?**

- Actividad del equipo en GitHub.

## **¿Hacia dónde va la información?**

- Plataforma de repositorio del proyecto (GitHub).

### **Ejemplo:**

```
git checkout -b tarea6-visitor
# implementar cambios
# push y crear PR
```