Analicemos cómo se comparan los traductores.

La Diferencia Fundamental: Intérprete vs. Traductor

- Un **intérprete** lee el código y *ejecuta* las acciones correspondientes de inmediato. El programa *es* la tortuga.
- Un **compilador** lee el código y lo *convierte* a otro lenguaje (en este caso, Python). El programa genera un archivo `.py` que luego se puede ejecutar.

Analogía: un intérprete es un traductor simultáneo en una conferencia, mientras que un compilador es alguien que toma un libro en un idioma y escribe un nuevo libro en otro.

¿Cómo Afecta a Cada Fase del Proyecto?

Las fases iniciales, en ambos traductores, que son las más importantes para el aprendizaje, **permanecen casi idénticas.**

1. Análisis Léxico (Scanner)

El scanner leerá cadenas en el pseudocódigo (`AVANZA 100`) y generará una secuencia de tokens: `[INSTRUCCION:"AVANZA", NUMERO:"100"]`.

2. Análisis Sintáctico (Parser)

El parser tomará los tokens y verificará que la gramática sea correcta. Su principal producto será un **Árbol de Sintaxis Abstracta (AST)**. La estructura de loop `REPITE 4 [ ... ]` por ejemplo, será convertida en un nodo `RepiteNode` con un valor de 4 y una lista de nodos hijos.

3. Tabla de Símbolos

- **¿Qué es?**: Es una estructura de datos (como un diccionario o una tabla hash) que almacena información sobre los identificadores (nombres de variables) que se han declarado.

- **Almacenamiento:** Cuando el parser encuentre `VAR LADO = 100`, la tabla de símbolos guardará que `LADO` existe. Más adelante, cuando encuentra la instrucción `AVANZA LADO`, en lugar de buscar un número, busca el identificador `LADO` en la tabla de símbolos. Obtiene el valor `100` y ejecuta la instrucción `AVANZA` con ese valor

- **Validación Semántica:** Si el código intenta usar `AVANZA DISTANCIA` sin haber declarado `DISTANCIA`, la tabla de símbolos te permitirá detectar este error semántico *antes* de generar el código Python. Esto es una parte fundamental del aprendizaje.

4. Gestor de Errores

Este es un módulo central que se comunica con todas las fases para reportar problemas de forma clara al usuario. Todos los errores léxicos, sintácticos y semánticos se detectarán de la misma manera y en las mismas fases antes de que se escriba una sola línea de Python.

- **Error Léxico:** Si encuentra un caracter inválido como `@` en el código. `Error: Caracter inesperado '@' en la línea 5.`

- **Error Sintáctico:** Detectado por el parser. `Error: Se esperaba un número después de 'AVANZA' en la línea 2.` o `Error: Falta ']' para cerrar el bloque 'REPITE' iniciado en la línea 8.`

- **Error Semántico:** Detectado durante la ejecución, a menudo con ayuda de la tabla de símbolos. `Error: La variable 'LADO' no ha sido definida. Línea 12.`

La Fase que Cambia:

Aquí es donde reside la diferencia, y es un cambio muy significativo.

En el Intérprete...	En el Compilador...
La fase final es un evaluador o ejecutor que recorre el AST.	La fase final es un generador de código que recorre el AST.

En el Intérprete...	En el Compilador...
Al visitar un AvanzaNode con valor 100, el evaluador llama a una función interna: mi_tortuga.avanzar(100);	Al visitar un AvanzaNode con valor 100, el generador de código crea un string: codigo_python += “t.forward(100)\n”
Al visitar un RepiteNode con valor 4, el evaluador ejecuta un bucle for en su propio código.	Al visitar un RepiteNode con valor 4, el generador escribe un bucle for de Python en el string de salida, incluyendo la indentación correcta: codigo_python += “for _ in range(4):\n” y luego procesa los nodos hijos con un nivel más de indentación.
Al encontrar VAR LADO = 100, el evaluador guarda 100 en la tabla de símbolos.	Al encontrar VAR LADO = 100, el generador escribe la línea de código Python: codigo_python += “LADO = 100\n”

Ejemplo

Input (pseudolenguaje):

```
# Dibuja un cuadrado usando una variable
INICIO
VAR LADO = 150
BAJAR_LAPIZ
REPITE 4 [
  AVANZA LADO
  GIRA_DERECHA 90
]
FIN
```

Output (El archivo `.py` que el compilador generará):

```
# Código generado automáticamente por el Traductor Tortuga
import turtle

# Configuración inicial
t = turtle.Turtle()
screen = turtle.Screen()
screen.title("Dibujo de Tortuga")
```

```
# --- Inicio del código traducido ---
LADO = 150
t.pendown()
for _ in range(4):
  t.forward(LADO)
  t.right(90)

# --- Fin del código traducido ---
# Mantener la ventana abierta
screen.mainloop()
```

Conclusión:

El enfoque del compilador:

1. **Refuerza la separación de fases:** La herramienta de análisis es completamente distinta de la ejecución del resultado.

2. **Introduce la generación de código:** Se aprende a transformar una estructura de datos (el AST) en código fuente válido.

3. **Produce un artefacto tangible:** Generar un archivo `.py` funcional y fácil de compartir.

4. **Simplifica la parte gráfica:** No se necesita gestionar el estado de la tortuga. Simplemente se genera el código que le dice a un módulo (`turtle`) cómo hacerlo.

Este proyecto, aunque reducido, mantiene toda la carga didáctica de un proyecto compilador real y la enfoca con los principios de los compiladores.