Universidad Católica “Nuestra Señora de la Asunción”

Facultad de Ciencias y Tecnología

Compiladores

Python to JavaScript Transpiler

Grupo:
Kevin March
Gabriel Gomez
José Marín

Profesores:
Luis Martinez
Ernst Heinrich Goossens

Asunción - Paraguay
2025

Índice

1. Introducción

Este documento describe el desarrollo de un *transpiler* que convierte código Python versión 3.6 en JavaScript ES6, utilizando herramientas de análisis léxico (*lex/flex*) y sintáctico (*yacc/bison*). Se explican las funcionalidades implementadas, las limitaciones detectadas, la forma de uso, la arquitectura del traductor, las decisiones de diseño, las técnicas de detección de errores y conclusiones generales.

2. Objetivos específicos

1. Analizar y convertir constructos básicos de Python (declaraciones, expresiones, control de flujo) a sintaxis equivalente en JavaScript.
2. Gestionar ámbitos de variables y generación de código organizado (declaraciones de funciones al inicio, cuerpo principal separado).
3. Detectar y reportar errores de sintaxis durante el proceso de parsing.
4. Documentar las limitaciones actuales del transpiler y proponer soluciones.
5. Facilitar el uso del traductor mediante un "How to" claro para compilación y ejecución.

3. Funcionamiento y Flujo de Construcción

1. **Analizador Léxico (scanner.l)**
    - Basado en la especificación de tokens de Python (palabras reservadas, operadores, literales).
    - Maneja indentación mediante una pila que emite tokens INDENT/DEDENT.
    - Ignora comentarios y espacios en blanco innecesarios.
2. **Analizador Sintáctico (parser.y)**
    - Gramática definida para sentencias simples y compuestas de Python.
    - Construye cadenas de texto (std::string) con fragmentos de código JavaScript.
    - Utiliza una tabla de símbolos (mapa de mapas) para controlar declaraciones de variables y evitar redeclaraciones.
    - Genera estructuras if, for, while, function, operadores y expresiones con equivalentes JS.
3. **Código Principal (main.cpp)**
    - Inicializa el ámbito global y gestiona argumentos de línea de comandos.

- Invoca el lexer y parser generados por Flex/Bison.
- Serializa las declaraciones de funciones y el cuerpo convertido a un archivo output.js.

La construcción del traductor se encaró de forma modular, abordando primero el análisis léxico, luego el sintáctico y finalmente la generación de código. Se eligió construir un transpiler de Python a JavaScript porque ambos lenguajes comparten ciertas similitudes de alto nivel (dinamismo, tipado flexible, estructuras de control parecidas), lo cual permite una traducción razonable sin perder la semántica. Esta elección también permitió abordar de manera práctica los desafíos del diseño de compiladores, enfocándose en la traducción entre lenguajes modernos y populares. También responde al objetivo académico de poner en práctica el uso de herramientas tradicionales en la construcción de compiladores.

El diseño de la gramática se basó en subconjuntos de Python que pudieran ser traducidos de manera razonable a JavaScript sin perder semántica. Se priorizó la claridad en la generación de código y la separación entre declaraciones de funciones y lógica principal, lo que facilitó tanto el desarrollo como la posterior depuración del traductor.

A lo largo del proceso se aplicaron pruebas constantes, ajustes incrementales a la gramática y mejoras progresivas en la gestión de ámbitos y estructuras. Este enfoque permitió identificar y superar obstáculos reales que surgen al pasar de la teoría a la práctica.

4. "How to" (Uso del Traductor)

1. **Descargar el código de la pagina de github:**
   https://github.com/Kevin-March/Python_to_Javascript
2. **Moverse a la carpeta src dentro del proyecto:**
   cd /src
3. **Generar analizadores:**
   bison -d -o parser.cpp parser.y
   flex -o scanner.cpp scanner.l
4. **Compilar el traductor:**
   g++ main.cpp parser.cpp scanner.cpp -o parser
5. **Ejecutar la conversión:**
   ./parser < archivo_entrada.py [archivo_salida.js]

Si no se especifica *archivo_salida.js*, se crea *output.js*.

5. Funcionalidades Implementadas

El desarrollo del transpiler de Python a JavaScript implicó implementar un conjunto de funcionalidades que abarcan aspectos léxicos, sintácticos y semánticos del lenguaje fuente, con el objetivo de producir un código JavaScript funcionalmente

equivalente. A continuación, se detalla cada funcionalidad implementada, los desafíos encontrados y las decisiones tomadas durante el proceso de diseño.

5.1 Comentarios y Espacios en Blanco

El analizador léxico está diseñado para detectar y omitir los comentarios de línea (#) y los comentarios multilínea (`""" ... """` y `''' ... '''`). La eliminación de estos elementos se realiza de forma silenciosa, ya que no afectan el flujo lógico del programa.

Además, se ignoran líneas completamente vacías y espacios en blanco irrelevantes. Sin embargo, los espacios iniciales de una línea se analizan cuidadosamente para determinar la indentación del bloque correspondiente, lo cual es crucial en Python.

5.2 Indentación y Estructura de Bloques

Uno de los aspectos distintivos de Python es su uso de la indentación como delimitador de bloques. Para manejar esto, el scanner implementa una pila de indentación que compara la cantidad de espacios/tabulaciones al comienzo de cada línea. Esto permite generar los tokens `INDENT` y `DEDENT`, que el parser utiliza para reconstruir jerárquicamente la estructura de bloques (condicionales, bucles, funciones, etc.).

Esta técnica emula el comportamiento del compilador de Python y permite traducir estructuras anidadas con fidelidad al código original.

5.3 Declaración y Asignación de Variables

El transpiler gestiona la declaración de variables utilizando una tabla de símbolos organizada por ámbito (`scope`). Cada vez que se encuentra una asignación a una variable, se verifica si ya fue declarada:

- Si no está registrada en la tabla de símbolos, se genera una declaración `let` en JavaScript.
- Si ya existe en el ámbito actual, se omite la redeclaración y solo se genera la asignación.

También se manejan operadores compuestos (`+=`, `-=`, etc.) y anotaciones de tipo opcionales (`x: int = 5`), las cuales son ignoradas en la salida porque JavaScript es débilmente tipado.

5.4 Expresiones Aritméticas, Booleanas y Comparativas

El parser reconoce y traduce correctamente expresiones aritméticas con operadores comunes (`+`, `-`, `*`, `/`, `%`, `**`, `//`), manteniendo la precedencia y asociatividad mediante reglas gramaticales y el uso de paréntesis.

- ** se traduce como ** (exponente en JS).
- // se traduce como `Math.floor(a / b)` para emular la división entera.
- Se manejan expresiones booleanas como `and`, `or`, `not` que se traducen a &&, ||, ! respectivamente.

Las comparaciones (==, !=, <, >, <=, >=) son convertidas a sus equivalentes estrictos en JS (===, !==, etc.) para evitar errores de coerción de tipos.

5.5 Condicionales (If, Elif, Else)

Se implementó una traducción completa de estructuras condicionales. El bloque `if` se convierte en `if (...) {}`, `elif` en `else if (...) {}`, y `else` en `else {}`.

El parser exige una sintaxis clara (uso de `:` y bloques con indentación válida), y organiza el código resultante respetando los niveles anidados.

5.6 Bucles (While, For)

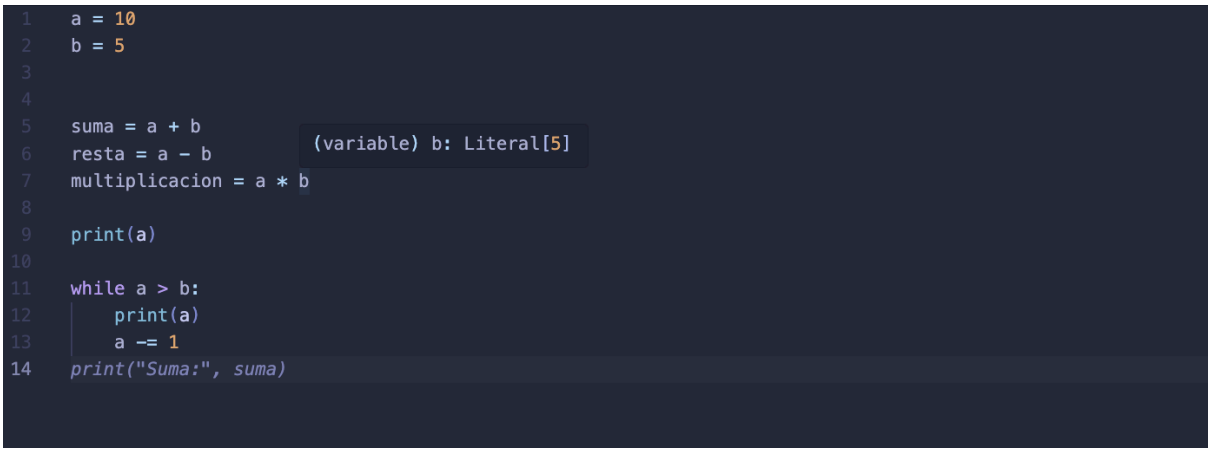
Bucle While:

La traducción del bucle `while` se implementa de forma directa como:

```
while condition:
    statements
```

se convierte en:

```
while ((condition)) {
    statements;
}
```



codigo while en python de test1.py

```javascript
// Código generado automáticamente de test1.py
// Traducción de Python a JavaScript


let a = 10;

let b = 5;

let suma = (a + b);

let resta = (a - b);

let multiplicacion = (a * b);

console.log(a)

while ((a > b)) {
    console.log(a)
    a = 1
}


```

genera este codigo en javascript, la salida fue salida1.js

El cuerpo del bucle se genera como un bloque, manteniendo la indentación traducida.

Bucle For:

El `for` admite dos formas principales:

1. `for x in range(n)`

```python
# Python
# Inicialización de una variable string
variable = "Hello World!!!"
# Cambiamos el tipo de variable
variable = 2024
# Inicialización de una variable int
numero = 10
# Asignación de un float a la variable int
numero = 10.567

print(variable)
print(numero)

for i in range(5):
    print(i)
```

se convierte en un bucle clásico:

```
for (let x = 0; x < n; x++) { ... }
```

```javascript
// Código generado automáticamente de test2.py
// Traducción de Python a JavaScript


let variable = "Hello World!!!";

variable = 2024

let numero = 10;

numero = 10.567

console.log(variable)

console.log(numero)

for (let i = 0; i < 5; i++) {
    console.log(i);
}

```

2. `for x in lista`

```javascript
for x in lista:
    x= x + 1
```

se traduce como:

`for (let x of lista) { ... }`

```javascript
// Código generado automáticamente de test6.py
// Traducción de Python a JavaScript


for (let x of lista) {
    x = (x + 1);
}


```

Esto permite recorrer secuencias como listas, manteniendo la lógica original de Python.

5.7 Funciones

El transpiler implementa el reconocimiento de funciones definidas con `def`. Las funciones son convertidas a funciones de JavaScript con `function`, respetando el nombre, parámetros y cuerpo.

Además:

- Los parámetros se agrupan en una lista y se insertan en la cabecera JS.
- Las funciones pueden incluir `return`, y su tipo de retorno (si está anotado) es ignorado en la salida.
- Todas las funciones generadas se agrupan al inicio del archivo JS, manteniendo orden lógico.

5.8 Listas y Estructuras de Datos

El transpiler traduce listas de Python (`[1, 2, 3]`) directamente a arreglos de JavaScript. También se habilita su uso como iterables en bucles `for...of`, y se respeta su sintaxis para indexación o acceso.

6. Limitaciones y Posibles Soluciones

1. **Ámbitos Global vs. Local**
    - *Causa:* JavaScript trata let en bloque, Python en función/glob.
    - *Solución:* Mantener pila de scopes y generar declaración var o const global según contexto.
2. **Tipado de Arrays y Listas**
    - *Causa:* Solo se traduce literal [...]; no inferencia de tipo.
    - *Solución:* Implementar análisis estático de contenido o anotaciones de tipo en comentarios JS.
3. **Unpacking** (a,b=c) No Soportado
    - *Causa:* Gramática no maneja asignaciones múltiples.
    - *Solución:* Extender reglas de expr_stmt para admitir tuplas y generar desestructuración JS [a,b]=c;.
4. **Comparaciones Encadenadas** (a < b < c) No Soportado
    - *Causa:* Gramática solo admite comparison comp_op expr simples.
    - *Solución:* Descomponer en operaciones lógicas encadenadas: (a < b && b < c).
5. **Funciones Anidadas y Closures**
    - *Causa:* No se mantiene functionDeclarationsStr por ámbito.

- *Solución:* Gestionar pila de funciones y generar closures o funciones anónimas.

6. **Manejo de Excepciones y try/except**
   - *Causa:* No hay reglas para try/except.
   - *Solución:* Añadir gramática y mapear a try { ... } catch(e) { ... }.
7. **Decoradores, Generators y Async/Await**
   - *Causa:* Características avanzadas omitidas.
   - *Solución:* Añadir gramática y traducir a macros o sintaxis ES6.

7. Técnicas de Detección de Errores

La solución más sencilla que se nos ocurrió fue primeramente mostrar en consola los pasos que estaba tomando el compilador, esto nos ayudó a debugear por si nuestra reglas

- Uso de yyerror para reportar errores de sintaxis con línea y token.

```
for i in range((5):
    print(i)
```

en el ejemplo hay disparidad de ()

```
Error de sintaxis: syntax error, unexpected COLON, expecting RPAREN or COMMA en la línea 1 (token: 282)
Error durante la traducción
```

otro ejemplo:

```
a="esta es una cadena"
z= 2*3
# Python - Función mal asignada
def = 42
```

utilización errónea de palabras reservadas.

```
[stmt_list_opt] agregando stmt
atom -> INTEGER: 2
atom_expr -> atom: 2
[power] atom_expr evaluado
[factor] power evaluado
[term] factor evaluado
atom -> INTEGER: 3
atom_expr -> atom: 3
[power] atom_expr evaluado
[factor] power evaluado
[term] Multiplicación detectada
[arith_expr] term evaluado
[expr] arith_expr evaluado
[comparison] expr evaluado
[not_test] comparison evaluado
[and_test] not_test evaluado
[or_test] and_test evaluado
[test] or_test evaluado
[expr_stmt] Asignación a variable: z
  Variable NO declarada previamente. Declarando...
[stmt_list_opt] agregando stmt
Error de sintaxis: syntax error, unexpected EQUALS, expecting IDENTIFIER en la línea 4 (token: 266)
Error durante la traducción
```

- Validación de indentación en el *scanner* con pila de niveles.

```
1    while true:
2    let i = 0;
3    if i == 5 :
4    print("error por falta de indentacion")
5
6
7
8
```

yyerror debe mostrar el error por falta de indentación

```
[stmt_list_opt] vacío
atom -> IDENTIFIER: true
Error de sintaxis: Variable no declarada: true en la línea 1 (token: 282)
atom_expr -> atom: true
[power] atom_expr evaluado
[factor] power evaluado
[term] factor evaluado
[arith_expr] term evaluado
[expr] arith_expr evaluado
[comparison] expr evaluado
[not_test] comparison evaluado
[and_test] not_test evaluado
[or_test] and_test evaluado
[test] or_test evaluado
Error de sintaxis: syntax error, unexpected IDENTIFIER, expecting INDENT en la línea 2 (token: 258)
Error durante la traducción
```

8. Conclusiones Generales

El transpiler demuestra que es posible automatizar la conversión básica de Python a JavaScript usando herramientas clásicas de compiladores. Sin embargo, las diferencias semánticas entre ambos lenguajes requieren un manejo cuidadoso de scopes, tipado dinámico, y características avanzadas como excepciones y generators. Las limitaciones actuales abren oportunidades para extender el proyecto con soporte para destructuring, comparaciones encadenadas, funciones asíncronas y un análisis de tipos más profundo.

En futuras versiones, se recomienda enriquecer la gramática, mejorar la tabla de símbolos para tipos y scopes, e implementar generar tests automáticos para verificar equivalencia funcional entre el código Python original y el JavaScript generado.

Este trabajo permitió aplicar de forma concreta los conceptos fundamentales del diseño de compiladores, tales como análisis léxico, sintáctico, manejo de ámbitos, construcción de árboles sintácticos y generación de código. Además, exigió un proceso iterativo de diseño, depuración y mejora que favoreció la comprensión profunda del funcionamiento interno de los lenguajes y de las herramientas utilizadas.