UNIVERSIDAD CATÓLICA
"NUESTRA SEÑORA DE LA ASUNCIÓN"

Facultad de Ciencias y Tecnología

Compiladores

Transpilador Python a C

Estudiante:

Oliver, Amparo

Profesores:

Luis Martinez
Ernst Heinrich Goossen

Asunción - Paraguay
Año 2024

Índice

1. Motivación

Python es un lenguaje de programación ampliamente utilizado, eficiente y fácil de aprender. En contraste, C es reconocido por ser riguroso y complejo. Me atrajo la idea de desarrollar un traductor de Python a C justamente debido a estas diferencias extremas, de poder identificar estas limitaciones en el proceso de traducción y tratar de entender cómo abordarlas. Además, como Python es de tipado dinámico y C de tipado estático, tenía claro desde el inicio ciertas limitaciones que encontraría en el proceso.

2. Cómo utilizar el traductor

Visita el siguiente enlace: https://github.com/amparooliver/Python_to_C_Transpiler

Clona el repositorio

```
git clone https://github.com/amparooliver/Python_to_C_Transpiler.git
```

Navegar a la carpeta v2

Accede a la carpeta v2 del repositorio clonado

```
cd v2
```

Limpiar archivos

Ejecutar el siguiente comando para limpiar archivos innecesarios

```
rm -f output.c parser parser.hpp parser.output scanner.cpp
```

Generación del Analizador Lexico y Sintactico

```
bison -d -o parser.cpp parser.y
flex -o scanner.cpp scanner.l
```

Compilación del Código

Compila el código utilizando g++

```
g++ main.cpp parser.cpp scanner.cpp -o parser
```

Prueba con Casos de Test

```
./parser < test_files/test1.py
```

3. Funcionalidades

3.1 Comentarios y Espacios en Blanco

El escáner (archivo lex scanner.l) es capaz de detectar comentarios en línea (#) y comentarios multilínea (""" ... """) de Python. **Para el propósito de este proyecto, simplemente ignoramos los comentarios, los espacios en blanco y las líneas en blanco, ya que no afectan la lógica del programa traducido.**

3.2 Declaración y Asignación de Variables

Primero, es fundamental entender cómo se maneja cada lenguaje en cuanto a variables y sus tipos para comprender ciertas decisiones que se tomaron.

Variables en Python

En Python, las declaraciones de variables no son necesarias, y las asignaciones son dinámicas. Esto significa que el tipo de una variable se determina en tiempo de ejecución y puede cambiar durante la ejecución del programa.

Tipos de datos numéricos

- **int**: Representa números enteros sin punto decimal.
- **float**: Representa números de punto flotante. Utiliza doble precisión (64 bits) para almacenar los valores.

Tipos de datos de cadenas

- **str:** Representa una cadena de caracteres.

Tipos de secuencia

- **Listas (arreglos o vectores)**: Representa una lista de elementos, que pueden ser de diferentes tipos de datos.

Tipo booleano

- **bool**: Representa valores booleanos: 'True' o 'False'.

Constantes en Python

Las constantes en Python no existen formalmente (simplemente por convención se escriben en mayúsculas, pero se tratan como cualquier variable). **Para el propósito de este proyecto, asumimos que las variables mayúsculas son constantes, es decir, inmutables.**

Variables en C

Tipos de datos numéricos

- **int**: Representa números enteros sin punto decimal.
- **float**: Representa números de punto flotante con precisión simple (usualmente 32 bits).
- **double**: Representa números de punto flotante con precisión doble (usualmente 64 bits).

Tipos de cadenas (Strings)

- **char[]**: Representa una cadena de caracteres como un array de 'char'.

Arrays

- **Array**: Representa una lista de elementos, que pueden ser de diferentes tipos de datos.

Tipo booleano

- **bool**: Representa valores booleanos: 'True' o 'False'.

Constantes en C

Una constante en C es un nombre asignado por el usuario a una ubicación en la memoria, cuyo valor no puede modificarse una vez declarado. Puedes declarar una constante en un programa en C de dos maneras: const int MAX_VALUE = 100; o #define MAX_VALUE 100. **Para el propósito de este proyecto, consideramos solo el primer tipo.**

¿Cómo maneja el transpilador la asignación de variables?

El transpilador mantiene una tabla de símbolos con el identificador y su tipo de dato. Si es la primera vez que encuentra el identificador, lo guardará en la tabla de símbolos con su tipo y realizará la declaración equivalente en C. **Por el momento no se agregan librerías necesarias como #include <stdbool.h>, #include <string.h>.**

```python
# Asignaciones validas
numero = 45
var1 = 1
var_float = 1.5
pi_double = 3.14159265359
varConTexto = "Hola!"
BoolVar = True
var_lista = [1, 2, 3, 4, 5] # Las listas no existen en C, se usan arrays
lista_con_letras = ['h', 'c', 'e'] # array de caracteres
CONSTANTE = 1234
```

Figura 1. Archivo test1.py

```c
#include <stdio.h>

int main() {

int numero = 45;
int var1 = 1;
float var_float = 1.5;
double pi_double = 3.14159265359;
char varConTexto[] = "Hola!";
bool BoolVar = true;
int var_lista[] = {1, 2, 3, 4, 5};
char lista_con_letras[] = {'h', 'c', 'e'};
const int CONSTANTE = 1234;

return 0;
}
```

Figura 2. Traducción generada en output.c

Si el transpilador detecta un identificador ya almacenado en la tabla de símbolos, simplemente realiza la asignación correspondiente si es válida en C. Dado que Python es un lenguaje de tipado dinámico, es válido tener una asignación inicial de un tipo, como '**`int`**', y posteriormente asignarle un valor de otro tipo, como '**`string`**'. Sin embargo, en C, una vez asignado un tipo de dato a una variable, este tipo de dato es *inmutable*.

La traducción de asignaciones de variables a valores/variables del mismo o distinto tipo numérico y booleano es posible, ya que en C el compilador se encarga de realizar truncamientos (de ser necesario) o interpretar valores numéricos asignados a una variable booleana, considerando cualquier valor distinto de cero como verdadero.

El transpilador, en este caso, realizará la traducción de manera literal. Si se asigna a una variable de tipo '**`int`**' algún valor '**`float`**' o una variable de tipo '**`float`**', la traducción será esa misma asignación, pero la interpretación posterior del compilador en C truncará los valores decimales. En el caso de las listas y strings, cualquier asignación dará un WARNING en la terminal y comentara esa asignación, como se puede observar en la Figura 4. La Figura 3 muestra las asignaciones válidas en Python y la Figura 4 muestra las traducciones que producirá el transpilador. **Este enfoque puede generar problemas en el flujo del código y pérdida de información.**

```python
# Declaraciones validas
a = 45
b = 1.5
c = True
var_string = "Hola!"
var_lista = [1, 2, 3, 4, 5] # Las listas no existen en C, se usan arrays

# Asignaciones a variables existentes
a = 40.5
b = False
c = a
var_string = 1
var_lista = 1
```

Figura 3. Tipado dinámico de Python.

```c
#include <stdio.h>

int main() {

int a = 45;
float b = 1.5;
bool c = true;
char var_string[] = "Hola!";
int var_lista[] = {1, 2, 3, 4, 5};
a = 40.5;
b = false;
c = a;
// var_string = 1; // Asignacion no valida en C, revisar si afecta el flow
// var_lista = 1; // Asignacion no valida en C, revisar si afecta el flow

return 0;
}
```

Figura 4. Traducción generada en output.c

3.2 Ciclos

While loop

En Python, el bucle '**`while`**' permite ejecutar repetitivamente un bloque de código mientras la condición sea verdadera. Durante la ejecución del bucle, se pueden utilizar estructuras de control como '**`break`**' para salir del bucle, interrumpiendo su ejecución.

```python
# Python While Loops
i = 1
a = 6
while i < a:
    i = i + 1
    if i == 3:
        break
```

Figura 5. Código Python de un While Loop

```c
#include <stdio.h>

int main() {

    int i = 1;
    int a = 6;
    while (i < a) {
        i = i + 1;
        if(i == 3) {
            break;
        }
    }

    return 0;
}
```

Figura 6. Traducción generada en output C.

Una limitación del transpilador es que no admite código inmediatamente después del break (aun estando dentro del while). Si es que el código continua, se debe dejar una línea en blanco entre el break y el siguiente statement. Es decir:

PERMITIDO	NO PERMITIDO	NO PERMITIDO

`# Python While Loops`
`i = 1`
`a = 6`
`while i < a:`
`    i = i + 1`
`    if i == 3:`
`        break`
 	
`i = i + 1`
	`# Python While Loops`
	`i = 1`
	`a = 6`
	`while i < a:`
	`    i = i + 1`
	`    if i == 3:`
	`        break`
	`    i = i + 1`
		`# Python While Loops`
		`i = 1`
		`a = 6`
		`while i < a:`
		`    i = i + 1`
		`    if i == 3:`
		`        break`
		
		`    i = i + 1`

For Loop

En Python, el bucle **`for`** proporciona varias formas de iterar sobre secuencias como listas, tuplas, cadenas de texto y rangos numéricos. **Para el propósito de este proyecto, se consideró solo la forma 'for i in range(x)', donde 'i' puede tener cualquier nombre de variable permitido y 'x' puede ser un identificador o un número entero definido.**

```
# Bucle for anidado para calcular productos

producto = 0
a = 2
for i in range(4):  # Itera sobre i = 1, 2, 3
    for j in range(a):  # Itera sobre j = 1, 2, 3
        producto = i * j  # Realiza la operación matemática
```

Figura 7. Ejemplo de For Anidado

```
#include <stdio.h>

int main() {

int producto = 0;
int a = 2;
for (int i = 0; i <  4; i++) {
for (int j = 0; j <  a; j++) {
producto = i * j;
}}
return 0;
}
```

Figura 8. Traducción generada en output.c

3.2 Condicionales

If Statement

En Python, la declaración **`if`** se utiliza para ejecutar un bloque de código si una condición es verdadera (True). Puede estar seguida opcionalmente por **`elif`** (abreviatura de "else if") y **`else`** para manejar múltiples condiciones o proporcionar un caso por defecto cuando la condición inicial no se cumple. **Una limitación del traductor es la necesidad de dejar una línea en blanco entre bloques condicionales o de insertar un comentario para marcar el fin correspondiente del bloque, como se muestra en la figura 9. Esto asegura que los bloques sean identificados correctamente. Mientras los bloques `if` y `else` permiten múltiples declaraciones, `elif` actualmente solo admite una (a revisar).**

```python
# Ejemplo if elif

edad = 25
puede_votar = False
if edad < 18:
    puede_votar = False

elif edad > 18:
    puede_votar = True

else:
    puede_votar = False
```

Figura 9. Condicional If Elif Else

```c
#include <stdio.h>

int main() {

int edad = 25;
bool puede_votar = false;
if(edad < 18) {
puede_votar = false;
} else if(edad > 18) {
puede_votar = true;
}
else {
puede_votar = false;
}
return 0;
}
```

Figura 10. Traducción generada en output.c

3.2 Estructuras anidadas

Debido a la forma en que el transpilador maneja las indentaciones y dedentaciones, las estructuras anidadas deben seguir ciertos requisitos para ser correctamente interpretadas por el transpilador. La dedentación debe estar marcada al menos con una declaración o un comentario, para que el transpilador pueda detectarla adecuadamente.

Por ejemplo, en la figura 11 se muestra cómo se debe escribir el código en Python para que el transpilador sea capaz de traducirlo correctamente, como se ve en la figura 12.

```python
# Inicialización de variables
i = 0
limite_externo = 3
limite_interno = 5
resultado = 1
stmt = 1

# Bucle while externo
while i < limite_externo:
    # Condición if dentro del while
    if i == 0:
        # Condición if dentro del if
        if i != 0:
            # Bucle for dentro del segundo if
            for j in range(limite_interno):
                if j == 5:
                    stmt = 2
                else:
                    stmt = 3
                # dedentación
            stmt2 = "O podria marcar la dedentacion con otro statement"
        # dedentación
    # Incrementar la variable del bucle while
    i = i + 1
# Resultado final almacenado en la lista
resultado_final = resultado
```

Figura 11. Cómo se deberían hacer las estructuras anidadas.

```c
#include <stdio.h>

int main() {

    int i = 0;
    int limite_externo = 3;
    int limite_interno = 5;
    int resultado = 1;
    int stmt = 1;

    while (i < limite_externo) {
        if(i == 0) {
            if(i != 0) {
                for (int j = 0; j <  limite_interno; j++) {
                    if(j == 5) {
                        stmt = 2;
                    } else {
                        stmt = 3;
                    }
                }
                char stmt2[] = "O podria marcar la dedentacion con otro statement";
            }
        }
        i = i + 1;
    }
    int resultado_final = resultado;

    return 0;
}
```

Figura 12. Traducción generada en output.c. (Las indentaciones del output fueron generadas manualmente para mayor comprensión).

4. Procedimientos y Funciones

En Python, los procedimientos y las funciones se declaran de la misma forma, utilizando la palabra clave '**`def`**'. La principal diferencia es que una función retorna un valor mediante la instrucción '**`return`**', mientras que un procedimiento no lo hace. Debido a la complejidad de tener que rastrear si una función tiene una instrucción '**`return`**'(después de haber ya procesado la información), **en este proyecto solo se implementaron procedimientos, que se traducen a '`void`' en C. Además, se decidió que todos los parámetros (si es que tiene) de los procedimientos se declararán como '`int`'. Esta decisión se basa en que, determinar el tipo de datos adecuado requeriría un análisis detallado del propósito de cada procedimiento y de cómo se utiliza dentro del código fuente original en Python.**

```python
def recipeMenu():
    a = 2
def conParametros(a, b):
    c = a + b

conParametros(2, 3)
recipeMenu()
```

Figura 13. Declaración y llamada de procedimientos.

```c
#include <stdio.h>

void recipeMenu();
void conParametros(int a, int b);

int main() {

// Se deberia trasladar esta declaracion fuera del main
void recipeMenu() {
    int a = 2;
}
// Se deberia trasladar esta declaracion fuera del main
void conParametros(int a, int b) {
    int c = a + b;
}
conParametros(2, 3);
recipeMenu();

return 0;
}
```

Figura 14. Traducción generada en output.c.

5. Dificultades encontradas

La mayor dificultad encontrada durante el desarrollo fue la correcta identificación de las indentaciones y dedentaciones. Como se mencionó a lo largo de la documentación, en ocasiones fue necesario marcar explícitamente la dedentación con instrucciones o

comentarios para asegurar que los bloques fueran interpretados correctamente. Además, la gestión del dinamismo de las variables en Python y su traducción a C planteó desafíos significativos.

No se llegó a implementar (aun):

- Arrays multidimensionales
- Switch
- Funciones

6. Deteccion y Recuperacion de Errores

Debido a la naturaleza dinámica de Python, la detección de errores sobre variables es casi nula. Un requisito impuesto para este proyecto es que las constantes no puedan modificar su valor una vez establecido, aunque Python lo permita. En caso de que esto ocurra, se mostrará un mensaje de error indicando la infracción, y la instrucción en cuestión será comentada como medida de manejo. Este enfoque puede ocasionar problemas en el flujo del programa.

Python Code:

```python
# Asignaciones validas
CONSTANTE_INT = 2
CONSTANTE_FLOAT = 40.5

# Cambio no valido
CONSTANTE_INT = 10000
```

Mensaje en la terminal:

```
Error: Modificacion no valida a constante CONSTANTE_INT. En linea: 7
```

Output.c:

```c
#include <stdio.h>

int main() {

const int CONSTANTE_INT = 2;
const float CONSTANTE_FLOAT = 40.5;
// CONSTANTE_INT = 10000; // Asignacion no valida en C, revisar si afecta el flujo

return 0;
}
```

Como se mencionó anteriormente, las asignaciones no válidas en C generarán un aviso, y el traductor comentará esa instrucción en el archivo output.c.

Python code:

```python
variable = [1, 2, 4]
# Ahora le asignamos otro valor
variable = "Ahora es un string en python"
```

Mensaje en la terminal:

```
WARNING: No se puede realizar la traduccion de esta asignacion en C
```

Output.c:

```c
#include <stdio.h>

int main() {

int variable[] = {1, 2, 4};
// variable = "Ahora es un string en python"; // Asignacion no valida en C, revisar si afecta el flujo

return 0;
}
```

Cualquier otro error que no tenga un manejo directo no generará un archivo **`output.c`**, y en la terminal se mostrará un mensaje de error de sintaxis.

Mensaje en la terminal:

```
Error de sintaxis en la línea 6
```