Tarea 3

Profesores: Andrés Abeliuk - Jocelyn Simmonds
Auxiliares: Carlos Antil - Blaz Korecic - Diego Salas - Lucas Torrealba

Instrucciones.-

- La tarea debe ser resuelta en **grupos de 2 - 3 integrantes**. No olviden especificar claramente sus nombres y secciones en el informe.
- Deberá entregar un único archivo en formato .zip, que deberá contener 2 archivos: `analizador.{py,java,cpp}`, e informe.pdf.
- Si hacen uso de ChatGPT, deben indicar donde lo ocuparon, y explicar cómo determinaron que la(s) respuestas(s) entregada(s) por ChatGPT es(son) correcta(s).

En esta tarea, le pedimos construir grafos de control de flujo (CFG) de programas especificados en un lenguaje simple de programación. Dado que el CFG de un programa representa los distintos caminos que pueden ser ejecutados por el programa de entrada, también les pediremos que ocupen estos grafos para comprobar ciertas propiedades acerca de los programas especificados en este lenguaje simple de programación.

Especificando los programas de entrada.-

Nuestro lenguaje simplificado se parece a Python y permite:

- Invocar funciones: `print(...)`, `f(x, y)`, etc.
- Definir variables:
  - Los nombres validos para una variable son `a`, `b`, ..., `z`.
  - Se puede asignar un valor constante a una variable: `x = 4`, `y = -6`, etc.
  - Pueden ocupar expresiones aritméticas simples para definir una variable: `y = x + 2`, `x = x + 1`, etc.
  - Pueden invocar una función para definir el valor de una variable: `z = f(x, y)`, etc.
- Alterar el flujo del programa:
  - Pueden bifurcar la ejecucion sobre una condición simple: `if`, `if/else`
  - Pueden iterar sobre una condición simple: `while`
  - Una condición simple puede comparar valores: `expr1 op expr2`, donde `expr1` y `expr2` son nombres de variables, constantes o llamados a funciones, y `op` es un operador relacional (`<`, `<=`, `==`, `>=`, `>`, `!=`).
  - Una condición simple también puede ser la invocación de una función que retorna un valor Booleano: `es_especial(x)`, etc.

Un programa de entrada se especifica en un archivo de texto, donde hay solo una instrucción por linea, y la indentación se ocupa para agrupar instrucciones. A continuación, se muestran ejemplos validos de programas de entrada:

1. Un programa que solo define una variable `x`, e imprime su valor:

```
x = 1
print(x)
```

2. Un programa que define 3 variables e imprime un valor:

```
x = 5
y = x + 2
z = f(x, y)
print(x + y + z)
```

3. Un programa que bifurca sobre una condición simple:

```
x = 0
if x > 5:
    y = 3
else:
    y = 4
```

4. Un programa que itera sobre una condición simple:

```
x = 0
while x < 10:
    print(x)
    x = x + 1

y = x + 3
```

Pueden combinar los ejemplos anteriores para crear programas más complejos de entrada.

Creando el CFG de un programa.-

Primero, vamos a definir los *bloques básicos*: un bloque básico es una secuencia de instrucciones de un programa que sabemos que se van a ejecutar todas juntas, una tras otra. Por ejemplo, en el segundo programa de ejemplo, las 4 instrucciones siempre se van a ejecutar juntas, así que podemos ocupar un único bloque básico para representar estas 4 instrucciones.

En el caso del tercer programa, hay tres bloques básicos: uno para las instrucciones `x = 0` y `x > 5` (siempre se ejecutan juntas), otro bloque para el caso verdadero (`y = 3`), y otro para el caso falso (`y = 4`).

Ahora podemos definir el CFG de un programa. Intuitivamente, el CFG es un grafo $G = (V, E)$, donde V es el conjunto de los bloques básicos del programa, y E modela los posibles caminos de ejecución del programa. O sea, $(u, u') \in E$ si las instrucciones del bloque u' se pueden ejecutar inmediatamente después de ejecutar las del bloque u. Agregaremos también un nodo especial a V, que representa el `fin` de la ejecución del programa. Todo camino de ejecución debe terminar en este nodo. La figura 1 muestra los CFG para los primeros tres ejemplos.

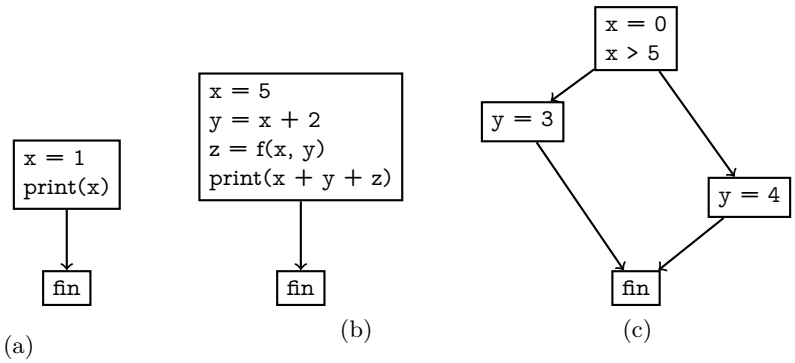


Figura 1: Los CFG correspondientes a los ejemplos 1, 2 y 3.

El caso del `while` es un poco más complejo, dado que introduce un ciclo en el grafo. Similar al caso del `if`, creamos bloques básicos separados para la condición del `while` y el código que se ejecuta dentro del `while` (esto se conoce como el *cuerpo* del `while`). A diferencia del `if`, la condición del `while` debe quedar en su propio bloque básico, dado que hay que volver a comprobar la condición después de cada vez que se ejecute el cuerpo del `while`. Entonces, también agregamos un arco desde el bloque que representa el cuerpo del `while` a la condición de este, para indicar que el cuerpo se puede ejecutar múltiples veces. Finalmente, también hay que agregar un arco desde el bloque que representa la condición del `while` al bloque que se ejecutara una vez que termine el `while` (o el nodo especial `fin` si no hay mas código a continuación del `while`. El CFG correspondiente al ejemplo 4 se muestra en la figura 2(a).

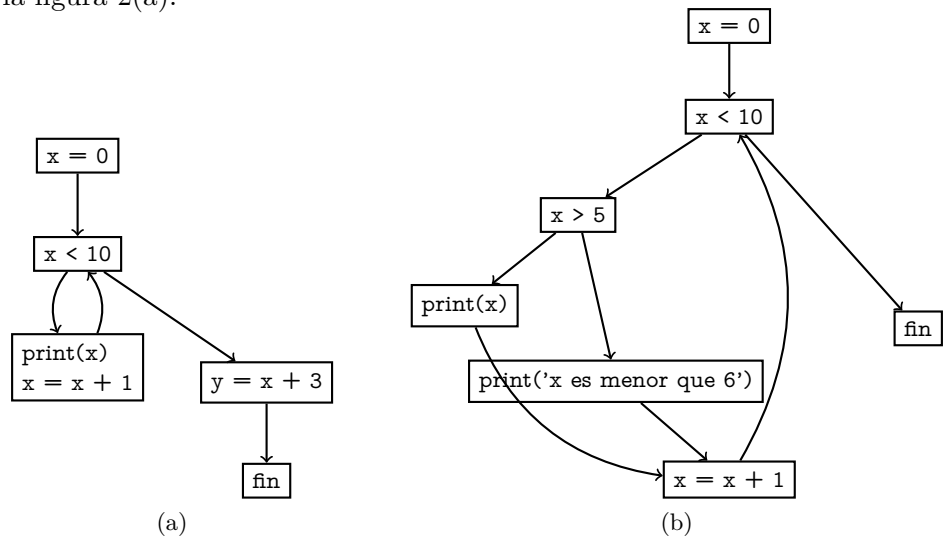


Figura 2: Los CFG correspondiente a los ejemplos 4 y 5.

Podemos crear programas más complejos en base a los ejemplos anteriores, como este ejemplo 5, cuyo CFG se muestra en la figura 2(b):

```
x = 0
while x < 10:
    if x > 5:
        print(x)
    else:
        print('x es menor que 6')
    x = x + 1
```

Analizando el CFG de un programa.-

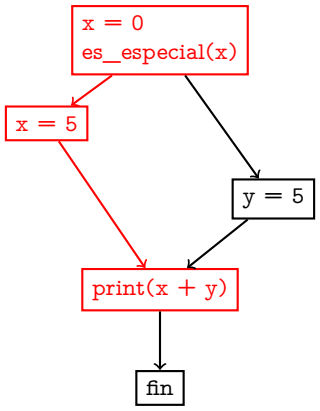
Una vez construido el CFG de un programa, deben implementar los siguientes dos análisis.

1. Encontrar variables posiblemente indefinidas

A veces tratamos de usar una variable antes de definirla, lo que es incorrecto. Por ejemplo, consideren el siguiente programa de entrada, donde podemos ejecutar la última instrucción sin haber definido la variable `y`, que es un error. Esto ocurre si la condición del `if` es verdadera:

```
x = 0
if es_especial(x):
    x = 5
else:
    y = 5
print(x + y)
```

A continuación mostramos el CFG de este programa, donde mostramos en rojo el camino que genera este error:



O sea, existe un camino desde el bloque básico inicial del CFG hasta el bloque correspondiente a `print(x + y)`, donde no se asigna valor a la variable `y`, lo que no es deseable.

2. Calcular la complejidad de un programa

McCabe propuso una formula para calcula la complejidad de un programa, en base a su CFG:

$$M = E - N + 2P$$

donde:

- E es el número de arcos del CFG
- N es el número de nodos del CFG
- P es el número de componentes conexos del CFG

M se conoce como la *complejidad ciclomática*, y corresponde al número de caminos independientes dentro de un fragmento de código.

Entrega.-

Deben implementar un programa llamado `analizador.{py,java,cpp}`, que reciba como entrada el nombre del archivo de texto donde se encuentra el programa de entrada a analizar, y que imprima la siguiente información, en el formato dado:

```
CFG
Nodos: 5
Arcos: 5
Componentes conexos: 1

Variables indefinidas
Variable: y
Camino: x = 0, es_especial(x) == True, x = 5, print(x + y)

Complejidad ciclomatica
2
```

En el caso de las variables indefinidas, deben listar todas las variables indefinidas que encuentren en el programa, y para cada una, deben indicar un camino donde esta indefinida. Si hay más de una variable indefinida, ordénelas de manera alfabética. Si una variable esta indefinida para más de un camino, basta con mostrar uno de estos.

También deben entregar un informe breve que explique a grandes rasgos las decisiones que tomaron al implementar la tarea (por ejemplo, como representaron los CFG), y que indique como ejecutar su código.