AUTLEN
Práctica 2 – Ejercicio 1
Abstract Syntax Trees (AST)

Ejercicio 1: Manipulando código en Python

Entre la batería de funciones que Python ofrece se encuentran algunas que permiten controlar y manipular la generación de código.

- La función compile, que permite compilar código fuente para obtener código compilado o un *abstract syntax tree* (AST).
- La función eval, que evalúa y retorna una expresión dado el código fuente o el compilado.
- La función exec, que ejecuta sentencias dado el código fuente o el compilado.

Ejercicio 1: Manipulando código en Python

Además, Python posee también varios módulos de utilidad para manipular su propio código:

- El módulo inspect nos permite inspeccionar y obtener información sobre los objetos básicos de Python. Por ejemplo, con la función getsource podemos inspeccionar el código fuente de un módulo, clase o función.
- El módulo ast tiene funciones para construir y manipular el árbol de sintaxis abstracta de la versión de Python que estemos ejecutando. Las funciones principales de este módulo son parse, que permite construir el árbol, y iter_fields, para recorrer los campos de un nodo. Además, usaremos las clases NodeVisitor y NodeTransformer, que permiten visitar y manipular el árbol usando el patrón de diseño Visitor.

Ejercicio 1. Apartado (a). Objetivo

La tarea a realizar en este apartado es la siguiente:

- Realizar un analizador sencillo de código, cuyo objetivo será contar el número de ocurrencias de números mágicos en el código (toda constante numérica distinta de 0, 1 y 1j -la unidad imaginaria-).
- Se debe crear un módulo "ast_utils.py" que contenga una clase "ASTMagicNumberDetector" que herede de NodeVisitor y almacene en un atributo llamado "magic_numbers" la cantidad observada de números mágicos.
- Se debe crear los métodos que se llamarán al visitar los distintos nodos del árbol en los que estemos interesados. En nuestro caso, estamos interesados únicamente en las constantes numéricas, que en versiones de Python anteriores a la 3.8 se definían como nodos de clase Num, y a partir de la versión 3.8 se definen como clase Constant (junto con otras constantes que antes tenían su propia clase, como Str).
- Para cada clase de nodo, NodeVisitor llama al método "visit_<class_name>", donde "class_name" es el nombre de la clase. Como queremos hacer nuestro código compatible con versiones de Python superiores a la 3.6, deberemos implementar los métodos correspondientes a las clases Num y Constant (y en este último caso comprobar que el valor de la constante es efectivamente numérico).

Ejercicio 1. Apartado (a). Esquema

```
# Clase a completar
class ASTMagicNumberDetector(ast.NodeVisitor):

    def __init__(self):
        self.magic_numbers = 0

    def _check_magic_number(self, number: complex) -> None:
        ... # usar isinstance(number, numbers.Number)

    def visit_Num(self, node: ast.Num) -> None:
        ... # usar node.n y _check_magic_number

    # Para Python >= 3.8
    def visit_Constant(self, node: ast.Constant) -> None:
        ... # usar node.value y _check_magic_number
```

Ejercicio 1. Apartado (a). Caso de prueba

```
# Ejemplo
import ast
import inspect
from ast_utils import ASTMagicNumberDetector

def my_fun(p):
    if p == 1:
        print(p + 1j)
    elif p == 5:
        print(0)
    else:
        print(p - 27.3 * 3j)

source = inspect.getsource(my_fun)
my_ast = ast.parse(source)

magic_detector = ASTMagicNumberDetector()
magic_detector.visit(my_ast)
print(magic_detector.magic_numbers)
# Debería dar 3
```

Ejercicio 1. Apartado (b). Objetivo

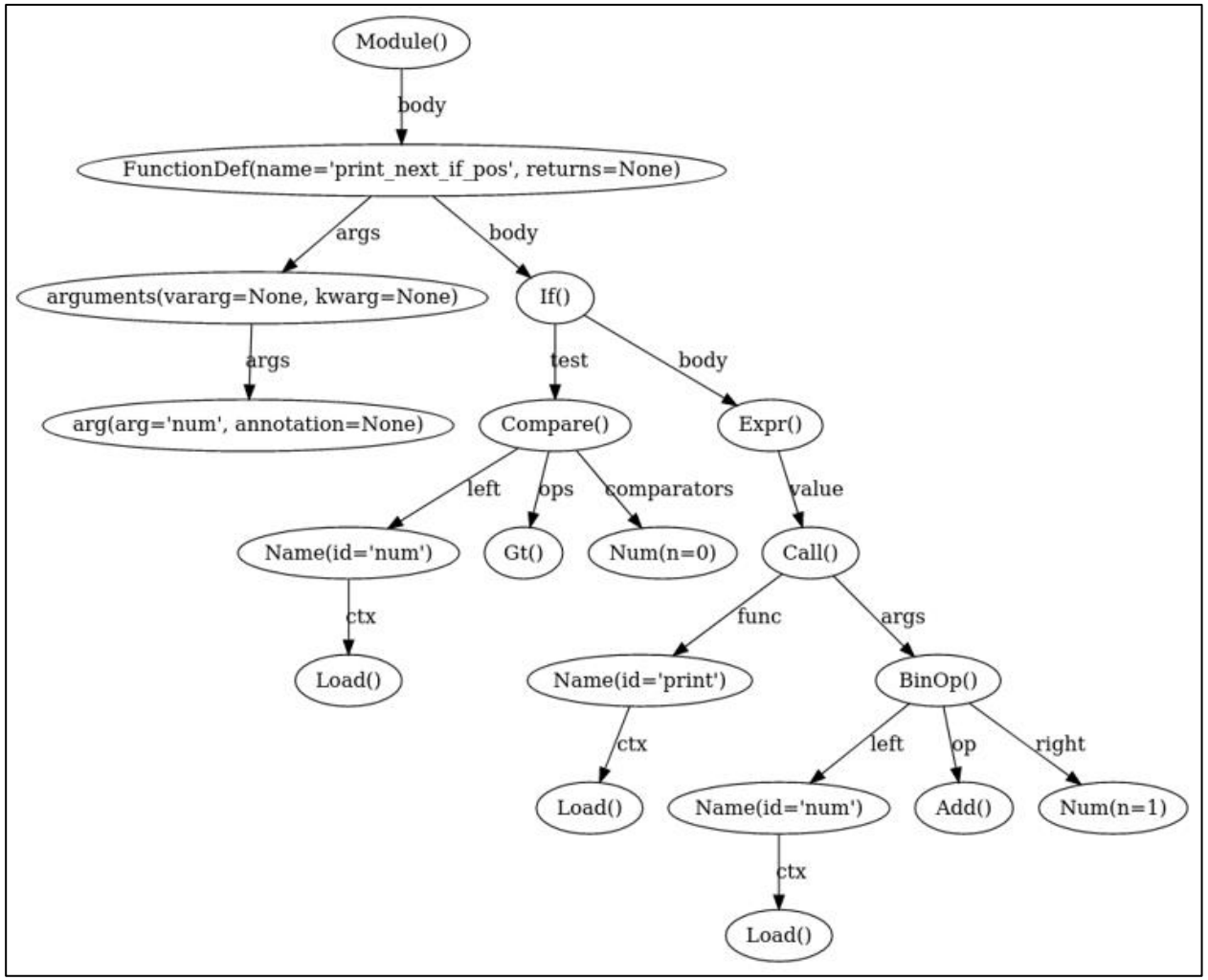
La tarea a realizar en este apartado es la siguiente:

- Usando la misma clase NodeVisitor, crear una subclase "ASTDotVisitor" en el mismo módulo, que imprima el AST en el formato *dot* de *Graphviz.*

Será útil para visualizar los siguientes ejercicios.

Aquí se muestra el *dot* para la siguiente función:

```
def print_next_if_pos(num):
    if num > 0:
        print(num + 1)
```



Ejercicio 1. Apartado (b). Explicación

- Como puede verse en el árbol, aquellos campos de un nodo que son a su vez nodos o listas de nodos se dibujan como descendientes del nodo, con el nombre del campo en la arista. Los campos que son objetos nativos de Python aparecen como parte del nombre entre paréntesis.
- En este caso queremos tener control sobre cómo NodeVisitor visita todos los nodos. Para ello, debemos reemplazar el método "generic_visit", que se llama cada vez que se visita un nodo que no tiene un método específico para su clase. Este método tiene la siguiente implementación por defecto:

```
def generic_visit(self, node):
    for field, value in iter_fields(node):
        if isinstance(value, list):
            for item in value:
                if isinstance(item, AST):
                    self.visit(item)
        elif isinstance(value, AST):
            self.visit(value)
```

- Por tanto, al reemplazarlo en la subclase será necesario que sigamos recorriendo los campos del nodo mediante "iter_fields" llamando a "visit" para los nodos hijos.
- Tened en cuenta que los campos pueden ser nodos, listas de nodos o valores primitivos de Python (en cuyo caso lo imprimiremos entre paréntesis dentro del mismo nodo, como en el ejemplo).
- Esta clase se ejecutará de forma similar al ejemplo anterior y deberá imprimir por pantalla el código *dot* generado durante la propia función "visit".

Ejercicio 1. Apartado (b). Esquema

```
# Clase a completar
class ASTDotVisitor(ast.NodeVisitor):

    def __init__(self) -> None:
        self.level = 0
        self.n_node = 0
        self.last_parent: Optional[int] = None
        self.last_field_name = ""

    def generic_visit(self, node: ast.AST) -> None:
        # usar un esquema similar al generic_visit de la clase padre para recorrer los hijos del nodo actual
        # obtener los nodos hijos (nodos AST y listas de nodos AST) del nodo actual
        # obtener los campos hijos (el resto) del nodo actual
        # procesar todos los campos hijos obtenidos del nodo actual (imprimir sus valores)
        # procesar todos los nodos hijos obtenidos del nodo actual (llamada a visit)
        …
```

Ejercicio 1. Apartado (b). Caso de prueba

- Por ejemplo, para nuestro caso, el código impreso debe ser similar a éste.

```
# Ejemplo
import ast
import inspect
from ast_utils import ASTDotVisitor

def print_next_if_pos(num):
    if num > 0:
        print(num + 1)

source = inspect.getsource(print_next_if_pos)
my_ast = ast.parse(source)

dot_visitor = ASTDotVisitor()
dot_visitor.visit(my_ast)
# Debería generar este texto dot
```

```
digraph {
s0[label="Module()"]
s1[label="FunctionDef(name='print_next_if_pos', returns=None)"]
s0 -> s1[label="body"]
s2[label="arguments(vararg=None, kwarg=None)"]
s1 -> s2[label="args"]
s3[label="arg(arg='num', annotation=None)"]
s2 -> s3[label="args"]
s4[label="If()"]
s1 -> s4[label="body"]
s5[label="Compare()"]
s4 -> s5[label="test"]
s6[label="Name(id='num')"]
s5 -> s6[label="left"]
s7[label="Load()"]
s6 -> s7[label="ctx"]
s8[label="Gt()"]
s5 -> s8[label="ops"]
s9[label="Num(n=0)"]
s5 -> s9[label="comparators"]
s10[label="Expr()"]
s4 -> s10[label="body"]
s11[label="Call()"]
s10 -> s11[label="value"]
s12[label="Name(id='print')"]
s11 -> s12[label="func"]
s13[label="Load()"]
s12 -> s13[label="ctx"]
s14[label="BinOp()"]
s11 -> s14[label="args"]
s15[label="Name(id='num')"]
s14 -> s15[label="left"]
s16[label="Load()"]
s15 -> s16[label="ctx"]
s17[label="Add()"]
s14 -> s17[label="op"]
s18[label="Num(n=1)"]
s14 -> s18[label="right"]
}
```

Ejercicio 1. Apartado (c). Objetivo

La tarea a realizar en este apartado es la siguiente:

- Usando la clase **NodeTransformer** crear una subclase **“ASTReplaceNum”** que reciba como argumento un número y reemplace todos los números del código por dicho número.

Su funcionamiento se puede comprobar mediante la función **“transform_code”** dada.

Esta función recibe una función de Python y un **NodeTransformer** y devuelve una nueva función usando el AST modificado.

```
def transform_code(f, transformer):
    f_ast = ast.parse(inspect.getsource(f))

    new_tree = ast.fix_missing_locations(transformer.visit(f_ast))

    old_code = f.__code__
    code = compile(new_tree, old_code.co_filename, 'exec')
    new_f = types.FunctionType(code.co_consts[0], f.__globals__)

    return new_f
```

Ejercicio 1. Apartado (c). Esquema

```
# Clase a completar
class ASTReplaceNum(ast.NodeTransformer):

    def __init__(self , number: complex):
        self.number = number

    def visit_Num(self, node: ast.Num) -> ast.AST :
        ... # devolver un nuevo nodo AST con self.number

    # Para Python >= 3.8
    def visit_Constant(self, node: ast.Constant) -> ast.AST :
        ...  # devolver un nuevo nodo AST con self.number si la constante es un número
```

Ejercicio 1. Apartado (c). Caso de prueba

```
# Ejemplo
from ast_utils import ASTReplaceNum

def my_fun(p):
    if p == 1:
        print(p + 1j)
    elif p == 5:
        print(0)
    else:
        print(p - 27.3 * 3j)

num_replacer = ASTReplaceNum(3)
new_fun = transform_code(my_fun, num_replacer)

new_fun(1)
# Debería imprimir -8

new_fun(3)
# Debería imprimir 6
```

Ejercicio 1. Apartado (d). Objetivo y Esquema

La tarea a realizar en este apartado es la siguiente:

- Simular la realización de un posible paso de optimización de un compilador. Para ello, se debe hacer una subclase “ASTRemoveConstantIf” de NodeTransformer que reemplace aquellos “if” del código cuya expresión a evaluar sea literalmente “True” o “False” por la parte del nodo “If” correspondiente.

En un compilador real, este paso podría ser una eliminación de ramas muertas después de simplificar la expresión de los if.

```
# Clase a completar
class ASTRemoveConstantIf(ast.NodeTransformer):

    def visit_If(self, node: ast.If) -> Union[ast.AST, List[ast.stmt]]:
        … # usar node.test, node.test.value, node.body y node.orelse
```

Ejercicio 1. Apartado (d). Caso de prueba

```
# Ejemplo
import ast
import inspect
from ast_utils import ASTRemoveConstantIf

def my_fun(p):
    if True:
        return 1
    else:
        return 0

source = inspect.getsource(my_fun)
my_ast = ast.parse(source)

if_remover = ASTRemoveConstantIf()
new_ast = if_remover.visit(my_ast)
```