Prácticas de Autómatas y Lenguajes. Curso 2021/22

Práctica 1: Autómatas finitos

Duración: 6 semanas.

Entrega: Lunes 15 de noviembre

Peso: 45% de la nota de prácticas

**En amarillo: relevante para el Ejercicio 1 (tiempo recomendado: 1 semana)

Descripción del enunciado:

En esta práctica trabajaremos con autómatas finitos, tanto deterministas como no deterministas. Se programará el algoritmo de aceptación de cadenas por parte de dichos autómatas, y se demostrarán mediante su implementación práctica los algoritmos de equivalencia y minimización de autómatas finitos.

Los objetivos de esta práctica serán:

- Entender y saber implementar el proceso y aceptación de símbolos y cadenas por parte de autómatas finitos no deterministas, de los que los deterministas son un caso particular.
- Entender la equivalencia entre expresiones regulares y autómatas finitos. Saber implementar la conversión de una expresión regular a un autómata finito no determinista equivalente.
- Entender la equivalencia entre autómatas finitos no deterministas y autómatas finitos deterministas. Saber implementar la transformación de un autómata finito no determinista en un autómata finito determinista equivalente.
- Entender y saber implementar la minimización de un autómata finito determinista en uno equivalente que no presente estados inaccesibles ni equivalentes.

Además, se pretenden lograr también los siguientes objetivos de programación:

- Entender y poder implementar código en el lenguaje de programación Python.
- Entender y saber implementar tests usando el módulo unittest, similar a JUnit.
- Entender y saber usar las anotaciones de tipos estáticos en Python, así como el analizador estático de tipos Mypy.

Se recuerda que los objetivos planteados deben ser adquiridos por **ambos** miembros de la pareja. En caso de realizarse una prueba y comprobar que este no es el caso, se podría suspender la práctica y exigir la entrega individual en la modalidad no presencial.

Descripción de los ficheros suministrados:

En esta práctica se proporcionan los siguientes ficheros:

- interfaces.py y re_parser_interfaces.py: Contienen clases abstractas de las que heredan las clases a implementar por el estudiante. Estas clases definen una API a la que el programador deberá adaptarse, además de tener ya implementada parte de la funcionalidad necesaria. El estudiante no debe modificar ni entregar estos ficheros.
- utils.py: Contiene funciones de utilidad para trabajar con autómatas finitos. El estudiante no debe modificar ni entregar este fichero.
- automaton.py: Contiene las clases básicas para definir un autómata finito no determinista, incluyendo sus estados y transiciones. El estudiante deberá añadir los métodos solicitados en la clase FiniteAutomaton. Opcionalmente podrá añadir nuevos atributos y métodos en las clases State, Transition y FiniteAutomaton, pero siempre respetando la interfaz suministrada en las clases abstractas de las que heredan.
- automaton_evaluator.py: Contiene una única clase que se usa para evaluar el autómata y comprobar si se aceptan o rechazan símbolos y cadenas. El estudiante deberá completar los métodos que faltan. Opcionalmente podrá añadir nuevos atributos y métodos, pero siempre respetando la interfaz suministrada.
- re_parser.py: Contiene una clase que se utiliza para convertir una expresión regular (usando la notación vista en clase) en un autómata finito. El estudiante deberá completar los métodos que faltan. Opcionalmente podrá añadir nuevos atributos y métodos, pero siempre respetando la interfaz suministrada.

Además de ello se proporcionan varios tests en formato unittest, útiles para comprobar que la funcionalidad implementada es correcta. El estudiante podrá añadir los test que crea necesarios.

Tipos y Mypy:

A diferencia de C, el lenguaje de programación Python no es un lenguaje de tipado estático, sino dinámico. Esto quiere decir que los tipos solo se comprueban en tiempo de ejecución. Además, al igual que muchos otros lenguajes de programación dinámicos, Python hace uso del llamado duck typing, permitiendo que las funciones y métodos acepten objetos arbitrarios que cumplan con la interfaz usada.

Aunque las mencionadas características facilitan el uso casual de Python, la iteración rápida de prototipos y la realización de proyectos de programación pequeños en corto plazo, la ausencia del chequeo de tipos exige mucha responsabilidad por parte del programador, así como la existencia de un amplio conjunto de tests para descartar la presencia de errores.

En aplicaciones grandes y/o complejas, la declaración explícita de tipos puede ayudar a encontrar errores que serían difíciles de hallar de otra manera. Por ello, las versiones recientes de Python permiten opcionalmente declarar los tipos de argumentos y retornos de funciones y métodos, atributos y variables. En la mayoría de los casos es suficiente con anotar los tipos de argumentos y retornos de funciones, ya que el resto usualmente puede ser inferido a partir de estos.

Las anotaciones no son usadas por Python en tiempo de ejecución. Sin embargo, hay programas como Mypy que permiten validar de forma estática que no hay errores de tipado.

El código que se proporciona está completamente anotado con tipos. Se recomienda descargar la herramienta mypy mediante pip:

```
pip install mypy
```

y realizar comprobaciones periódicamente ejecutando el siguiente comando:

```
mypy --strict --strict-equality <ruta_del_proyecto>
```

Además, muchos entornos de desarrollo integrado (IDEs) tienen integración con Mypy, pudiendo mostrar los errores en la línea en la que se producen. Muchos IDEs también son capaces de utilizar las anotaciones de tipos para realizar las recomendaciones de autocompletado de atributos y métodos.

Ejercicio 1: Evaluación del autómata (2 puntos):

Como primer paso el estudiante debe leer y entender las clases State, Transition, FiniteAutomaton y FiniteAutomatonEvaluator, así como los atributos y métodos que heredan de sus respectivas superclases.

Se debe completar el fichero automaton_evaluator.py para permitir la validación de cadenas usando un autómata finito. Se deben implementar los siguientes métodos de la clase FiniteAutomatonEvaluator:

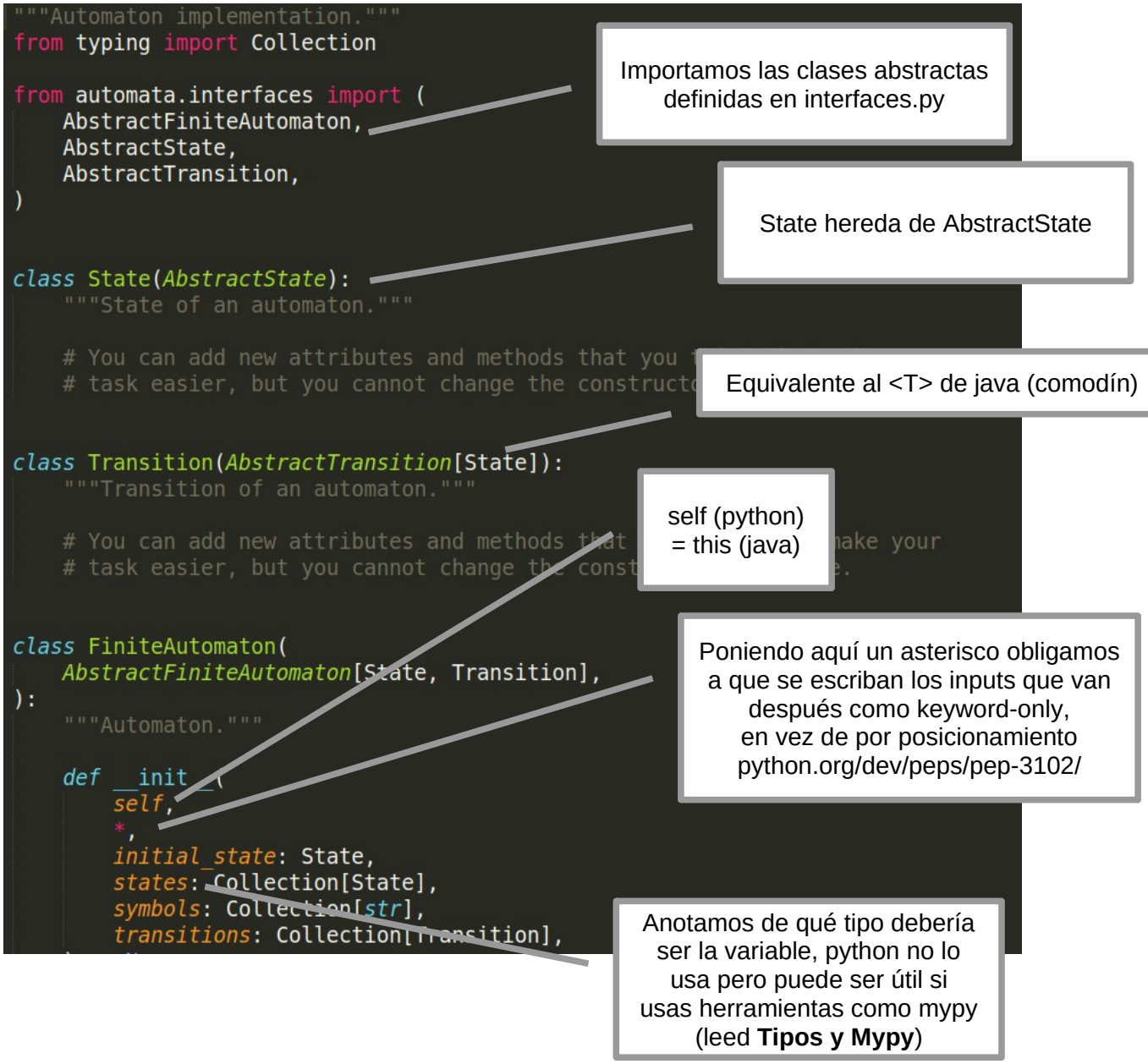
- process_symbol: Procesa un símbolo de la cadena (y cualquier número de transiciones lambdas inmediatamente después, mediante la llamada a _complete_lambdas).
- _complete_lambdas: Añade al conjunto de estados pasado como argumento todos los estados que sean alcanzables mediante un número arbitrario de transiciones lambda.
- is_accepting: Indica si la cadena que se ha procesado hasta el momento se acepta o no.

Al completar esta función, los tests en test_evaluator deberían pasar. Se

recomienda a los estudiantes que intenten también crear sus propios tests para comprobar todos los casos.

Notas sobre python:

automaton.py



automaton_evaluator.py

```python
"""Evaluation of automata."""
from typing import Set

from automata.automaton import FiniteAutomaton, State
from automata.interfaces import AbstractFiniteAutomatonEvalu


class FiniteAutomatonEvaluator(
    AbstractFiniteAutomatonEvaluator[FiniteAutomaton, State],
):
    """Evaluator of an automaton."""

    def process_symbol(self, symbol: str) -> None:

        raise NotImplementedError("This method must be implemented.")

    def _complete_lambdas(self, set_to_complete: Set[State]) -> None:

        raise NotImplementedError("This method must be implemented.")

    def is_accepting(self) -> bool:

        d must be implemented.")
```

Anotamos de qué tipo debería ser lo que devuelva este método, pero igual que los " : ", python no lo necesita para ejecutar

No debería devolver nada

Los métodos de las clases de python no pueden ser , privadas así que se usa el "_" delante del nombre para indicar que este método está pensado para un uso interno

Debería devolver **True** ó **False**

En test_evaluator.py

```python
class TestEvaluatorFixed(TestEvaluatorBase):
    """Test for a fixed string."""

    def _create_automata(self) -> FiniteAutomaton:

        description = """
        Automaton:
            Symbols: Helo

            Empty
            H
            He
            Hel
            Hell
            Hello final

            --> Empty
            Empty -H-> H
            H -e-> He
            He -l-> Hel
            Hel -l-> Hell
            Hell -o-> Hello
        """

        return AutomataFormat.read(description)

    def test_fixed(self) -> None:
        """Test for a fixed string."""
        self._check_accept("Hello", should_accept=True)
        self._check_accept("Helloo", should_accept=False)
        self._check_accept("Hell", should_accept=False)
```

Definimos símbolos que acepta nuestro autómata

Estados posibles

Estado de aceptación

Estado inicial

Transiciones según el estado inicial el siguiente símbolo, y estado final
Si no hay símbolo, es una transición λ

En utils.py
lee la cadena "description" y crea un autómata usando **expresiones regulares ;)**
Merece la pena echarle un ojo por curiosidad

Tests: palabras y resultado

Tests provistos:

- TestEvaluatorFixed: Autómata que sólo acepta la cadena “Hello”
- TestEvaluatorLambdas: Sólo acepta la cadena vacía
- TestEvaluatorNumber: Acepta números con un formato estándar.

Nota: ¿qué implementaciones son más eficientes en el procesamiento de símbolos en el caso de un autómata no determinista, que además sirva para el caso de los deterministas sin necesidad de crear dos códigos independientes?

Ejercicio 2: Conversión de expresiones regulares en autómatas finitos (2 puntos):

En este ejercicio se demostrará de forma práctica que para toda expresión regular existe un autómata finito equivalente.

Los estudiantes deben discutir en clase cómo se realizaría la conversión en autómata de cada uno de los elementos que conforman una expresión regular (operaciones y símbolos). Una vez puesto en común, los estudiantes deberán implementar en la clase REParser del fichero re_parser.py los siguientes métodos:

- _create_automaton_empty: Crea el autómata que acepta el lenguaje vacío.
- _create_automaton_lambda: Crea el autómata que acepta la cadena vacía.
- _create_automaton_symbol: Crea el autómata que acepta un símbolo.
- _create_automaton_star: Crea el autómata para calcular la estrella de Kleene de un autómata dado.
- _create_automaton_union: Crea el autómata para calcular la unión de dos autómatas dados.
- _create_automaton_concat: Crea el autómata para calcular la concatenación de dos autómatas dados.

Ejercicio 3: Conversión de autómatas finitos no deterministas en deterministas (3 puntos):

El profesor explicará en clase el algoritmo de conversión de autómata no determinista en determinista.

Los estudiantes deben implementar en la clase FiniteAutomaton el método to_deterministic, que realiza dicha conversión.

Ejercicio 4: Simplificación de autómatas finitos deterministas (3 puntos):

El profesor explicará en clase el algoritmo de minimización de autómatas deterministas.

Los estudiantes deben implementar en la clase FiniteAutomaton el método to_minimized, que realiza dicha minimización.

Ejercicio opcional (0.5 puntos, a añadir sobre la nota final de prácticas):

Diseña un conjunto exhaustivo de tests (mínimo 5 casos de tests para cada ejercicio, cada caso probando una funcionalidad concreta) para los ejercicios 3 y 4.

Los docstrings de cada caso de test deben incluir una descripción detallada de qué se pretende probar en dicho caso y por qué se ha seleccionado. Se deben incluir las imágenes generadas con dot (usando la función write_dot de utils.py) del autómata original y el transformado como parte de la entrega.

Planificación:

Ejercicio 1	Semana 1
Ejercicio 2	Semana 2
Ejercicio 3	Semanas 3 y 4
Ejercicio 4	Semanas 5 y 6

Normas de entrega:

La entrega la realizará sólo uno de los miembros de la pareja a través de la tarea disponible en Moodle.

Sólo se deben entregar los ficheros *automaton.py*, *automaton_evaluator.py* y *re_parser.py* además de los ficheros de tests e imágenes requeridas en caso de hacer el ejercicio opcional.

-