Prácticas de Autómatas y Lenguajes. Curso 2022/23

Práctica 2: Análisis Sintáctico

Duración: 6 semanas.

Entrega: Lunes 19 de diciembre, cada grupo antes de su clase.

Peso: 45% de la nota de prácticas.

Descripción del enunciado:

En esta práctica estudiaremos distintos conceptos relacionados con el análisis sintáctico de un lenguaje independiente del contexto. Empezaremos con algunos ejercicios sencillos en los que utilizaremos árboles sintácticos reales para el lenguaje de programación Python. A continuación implementaremos nuestro propio analizador sintáctico para gramáticas LL(1).

Los objetivos de la práctica son:

- Saber interpretar y manipular el árbol sintáctico de una gramática real a través del módulo ast de python.
- Calcular los conjuntos *primero* y *siguiente* de cada uno de los símbolos no terminales de una gramática.
- Construir la tabla de análisis para un analizador sintáctico descendente LL(1) a partir de los conjuntos *primero* y *siguiente* de la gramática.
- Entender y programar el algoritmo de análisis sintáctico descendente LL(1) a partir de la descripción de la tabla de análisis.

Además, se seguirá profundizando en los objetivos de programación utilizando el lenguaje Python planteados en la práctica 1.

Se recuerda que los objetivos de aprendizaje 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:

Se facilitan los siguientes ficheros para los ejercicios 2-5 (estarán disponibles en moodle la semana del 21 de noviembre):

- *grammar.py*: Contiene, entre otras, las clases `Grammar` y `LL1Table` que deberán ser modificadas para añadir los métodos pedidos en los ejercicios 2-5.

- *utils.py*: Contiene funciones de utilidad para, por ejemplo, leer una gramática a partir de su descripción en forma de cadena o imprimir la tabla de análisis. No se debe modificar ni entregar este fichero.
- *test_analyze.py*, *test_first.py* y *test_follow.py*: Contienen diversos tests de ejemplo en formato unittest, útiles como punto de partida para desarrollar nuevos tests que permitan comprobar que la funcionalidad implementada es correcta.

Ejercicio 1: Manipulando código en Python (3 puntos):

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.

Además, Python posee 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, e iter_fields, que se puede usar 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.

Las tareas a realizar en este ejercicio son las siguientes:

Apartado (a): Para empezar simularemos la realización de un analizador sencillo de código, cuyo objetivo será calcular el máximo nivel de anidamiento con bucles `for` en un fragmento de código. Para ello se deberá crear un módulo con nombre *ast_utils.py* que contenga la clase `ASTNestedForCounter`. Esta clase extenderá `NodeVisitor` e implementará los dos métodos siguientes:

- `generic_visit(self, node)`: Este método se invoca cada vez que se visita un nodo del AST que no tiene definido un método específico para su clase. Debe devolver el máximo nivel de anidamiento de bucles `for` a partir de ese nodo. Como ayuda, se muestra a continuación la implementación por defecto (en la clase `NodeVisitor`). Nótese que el algoritmo recorre todos

los campos del nodo (que pueden ser nodos, listas de nodos o valores primitivos de Python) visitándolos si son también nodos (`AST`).

```
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)
```

- `visit_For(self, node)`: Este método se invoca cada vez que se visita un nodo de tipo `for`. Debe devolver el resultado de añadir una unidad al máximo nivel de anidamiento de bucles `for` a partir de ese nodo.

El siguiente código muestra un ejemplo de uso de la clase implementada:

```
import ast
import inspect
from ast_utils import ASTNestedForCounter

def fun1(p):
    for a in [10, 20, 30]:
        print(a)
    for x in range(10):
        print(x)

def fun2(p):
    for a in [10, 20, 30]:
        print(a)
    for i in [10, 20, 30]:
        for j in [10, 20, 30]:
            for k in [10, 20, 30]:
                print(i)

def main() -> None:
    counter = ASTNestedForCounter()

    source = inspect.getsource(fun1)
    my_ast = ast.parse(source)
    print("Maximum number of nested for loops:", counter.visit(my_ast))
    # Should print 1

    source = inspect.getsource(fun2)
    my_ast = ast.parse(source)
    print("Maximum number of nested for loops:", counter.visit(my_ast))
    # Should print 3
```

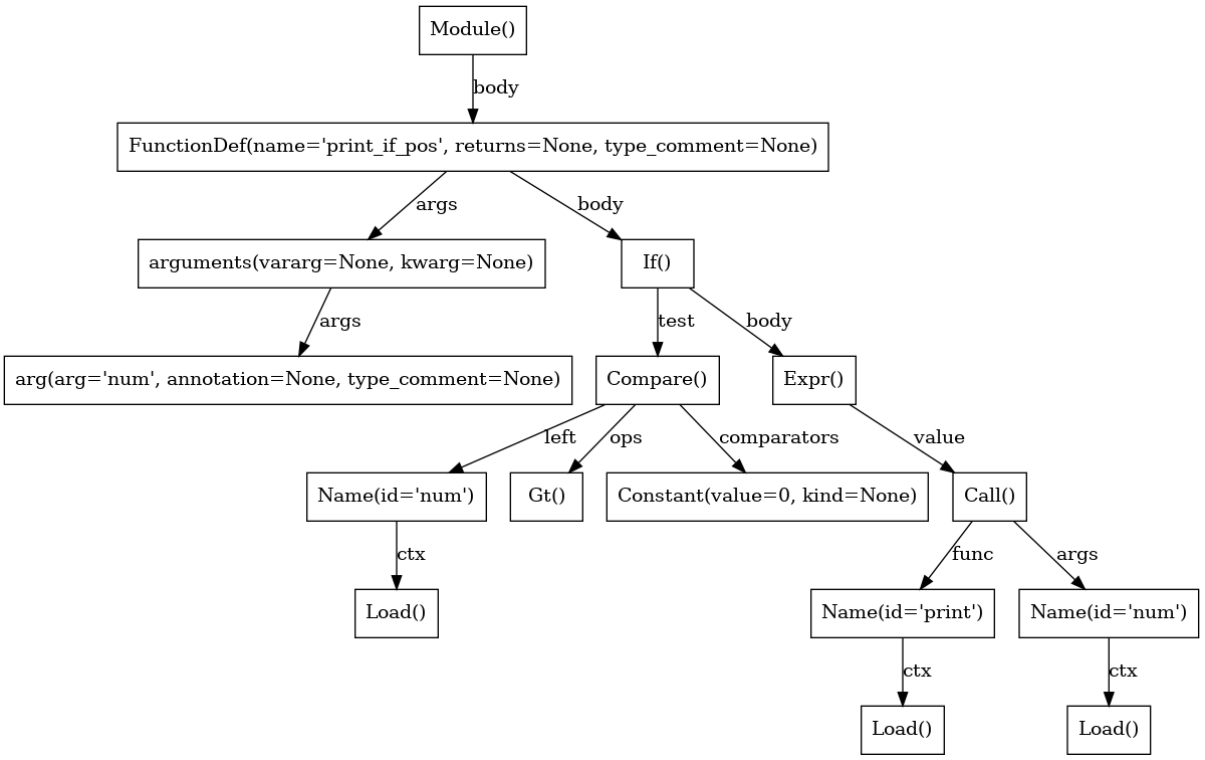
Apartado (b): Partiendo de la misma clase `NodeVisitor` crea, en *ast_utils.py*, la subclase `ASTDotVisitor` que imprima en la salida estándar el AST en el formato dot de Graphviz (será útil para visualizar los ejercicios siguientes). Por ejemplo, para la siguiente función

```
def print_if_pos(num):
    if num > 0:
        print(num)
```

se debería generar esta descripción:

```
digraph {
s0[label="Module()", shape=box]
s1[label="FunctionDef(name='print_if_pos', returns=None,
type_comment=None)", shape=box]
s0 -> s1[label="body"]
s2[label="arguments(vararg=None, kwarg=None)", shape=box]
s1 -> s2[label="args"]
s3[label="arg(arg='num', annotation=None, type_comment=None)", shape=box]
s2 -> s3[label="args"]
s4[label="If()", shape=box]
s1 -> s4[label="body"]
s5[label="Compare()", shape=box]
s4 -> s5[label="test"]
s6[label="Name(id='num')", shape=box]
s5 -> s6[label="left"]
s7[label="Load()", shape=box]
s6 -> s7[label="ctx"]
s8[label="Gt()", shape=box]
s5 -> s8[label="ops"]
s9[label="Constant(value=0, kind=None)", shape=box]
s5 -> s9[label="comparators"]
s10[label="Expr()", shape=box]
s4 -> s10[label="body"]
s11[label="Call()", shape=box]
s10 -> s11[label="value"]
s12[label="Name(id='print')", shape=box]
s11 -> s12[label="func"]
s13[label="Load()", shape=box]
s12 -> s13[label="ctx"]
s14[label="Name(id='num')", shape=box]
s11 -> s14[label="args"]
s15[label="Load()", shape=box]
s14 -> s15[label="ctx"]
}
```

Que, a su vez, generaría el siguiente gráfico:



Nótese que 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 en cambio como parte del nombre (entre paréntesis).

Apartado (c): Usando la clase `NodeTransformer` crea, en *ast_utils.py*, la subclase `ASTReplaceVar`. Los argumentos del constructor son el nombre de una variable y un AST. Al invocar al método `visit`, se deben reemplazar todas las apariciones de la variable en un contexto de tipo `Load` por ~~la constante 0~~ el AST (echa un vistazo a cómo se manejan las variables en el AST).

La diferencia entre las clases `NodeTransformer` y `NodeVisitor` es que la primera espera que los métodos implementados devuelvan un nodo que reemplace al nodo visitado. En caso de que no queramos reemplazar el nodo visitado bastará con devolver el propio nodo. Por tanto, el método `visit` modificará el AST original y retornará dicho AST modificado.

El código siguiente muestra un ejemplo, en el que el AST que reemplaza a la variable “num” es la constante 0:

```
import ast
import inspect
from ast_utils import ASTReplaceVar

def fun3(x):
    num = x + 2
```

```
    if num > 0:
        print(num)

source = inspect.getsource(fun3)
my_ast = ast.parse(source)
print(source)
repl = ASTReplaceVar("num", ast.Constant(0))
repl.visit(my_ast)
print(ast.unparse(my_ast))
```

La salida esperada es esta:

```
def fun3(x):
    num = x + 2
    if num > 0:
        print(num)

def fun3(x):
    num = x + 2
    if 0 > 0:
        print(0)
```

Nótese que la primera aparición de la variable `num` no se ha reemplazado por 0, pues se está asignando su valor (contexto `Store`).

Nota: el método `ast.unparse` sólo está disponible a partir de la versión 3.9 de python.

Apartado (d): Finalmente simularemos la realización de un posible paso de optimización de un compilador. Para ello se debe programar una subclase de `NodeTransformer`, con nombre `ASTUnroll`, que realice loop unrolling con todos los bucles `for` en los que el campo *target* sea una variable (`Name`) y el campo *iter* sea una lista.

El siguiente código muestra un ejemplo:

```
import ast
import inspect
from ast_utils import ASTUnroll

def fun4(p):
    for a in [10, 20, 30]:
        print(a)
        for b in [1, 2, 3]:
            print(2*a+b)
    for x in range(10):
        print(x)
```

```
    for i, x in enumerate([10, 20, 30]):
        print(i, x)

unroll = ASTUnroll()
source = inspect.getsource(fun4)
my_ast = ast.parse(source)
unroll.visit(my_ast)
print(ast.unparse(my_ast))
```

Esta es la salida esperada:

```
def fun4(p):
    print(10)
    print(2 * 10 + 1)
    print(2 * 10 + 2)
    print(2 * 10 + 3)
    print(20)
    print(2 * 20 + 1)
    print(2 * 20 + 2)
    print(2 * 20 + 3)
    print(30)
    print(2 * 30 + 1)
    print(2 * 30 + 2)
    print(2 * 30 + 3)
    for x in range(10):
        print(x)
    for (i, x) in enumerate([10, 20, 30]):
        print(i, x)
```

Ejercicio 2: Análisis sintáctico descendente (2 puntos):

En este ejercicio se implementará el algoritmo de análisis sintáctico descendente LL(1) a partir de una tabla de análisis dada. Para ello es necesario completar el código del método `analyze` de la clase `LL1Table` en el fichero `grammar.py`. El método recibe la cadena a analizar, `input_string`, y el axioma o símbolo inicial, `start`. Debe devolver un árbol de derivación (que puede estar vacío si no se realiza el ejercicio opcional) si la cadena es sintácticamente correcta de acuerdo a la tabla de análisis, o generar una excepción de tipo `SyntaxError` si no lo es.

En el fichero `test_analyze.py` se facilitan algunos tests. Recordad no obstante que es vuestra responsabilidad realizar tests adicionales para comprobar que el código funciona de manera correcta.

Ejercicio 3: Cálculo del conjunto *primero* (2 puntos):

Se debe completar, en la clase `Grammar` del fichero `grammar.py`, el código del método `compute_first`, que calcula el conjunto *primero* para una determinada cadena *w* recibida como argumento. Esta cadena debe estar formada únicamente por símbolos terminales y no terminales de la gramática. En caso de que no sea así el método deberá lanzar una excepción del tipo `ValueError`. El método debe devolver un conjunto con todos los símbolos contenidos en *primero(w)*, siendo cada símbolo una cadena con un único carácter, o una cadena vacía si $\lambda \in$ *primero(w)*.

En el fichero `test_first.py` se facilitan algunos tests. Recordad no obstante que es vuestra responsabilidad realizar tests adicionales para comprobar que el código funciona de manera correcta.

Ejercicio 4: Cálculo del conjunto *siguiente* (2 puntos):

Se debe completar, en la clase `Grammar` del fichero `grammar.py`, el código del método `compute_follow`, que calcula el conjunto *siguiente* para un determinado símbolo no terminal *X* recibido como argumento. El método debe generar una excepción del tipo `ValueError` si el símbolo recibido no pertenece al conjunto de símbolos no terminales de la gramática. En caso contrario el método debe devolver un conjunto con todos los símbolos contenidos en *siguiente(X)*, siendo cada símbolo una cadena con un único carácter, que puede incluir al carácter de fin de cadena `'$'`.

En el fichero `test_follow.py` se facilitan algunos tests. Recordad no obstante que es vuestra responsabilidad realizar tests adicionales para comprobar que el código funciona de manera correcta.

Ejercicio 5: Cálculo de la tabla de análisis LL(1) (1 punto):

Completad, en la clase `Grammar` del fichero `grammar.py`, el código del método `get_ll1_table`, que calcula la tabla de análisis LL(1) para la gramática. El método debe devolver un objeto de la clase `LL1Table`, o `None` si la gramática no es LL(1) (es decir, si hay varias partes derechas en una misma casilla de la tabla).

En el fichero `test_analyze.py` se facilitan algunos tests. Recordad no obstante que es vuestra responsabilidad realizar tests adicionales para comprobar que el código funciona de manera correcta.

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

Modificad el código del método `analyze` de la clase `LL1Table` (ejercicio 2) para que devuelva el árbol de derivación cuando la cadena a analizar sea sintácticamente correcta. El árbol de derivación debe ser un objeto de la clase `ParseTree`.

En el fichero `test_analyze.py` se facilitan algunos tests. Recordad no obstante que es vuestra responsabilidad realizar tests adicionales para comprobar que el código funciona de manera correcta.

Planificación:

Ejercicio 1	Semanas 1 y 2
Ejercicio 2	Semana 3
Ejercicio 3	Semana 4
Ejercicio 4	Semana 5
Ejercicio 5	Semana 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 *ast_utils.py*, con el código desarrollado para el ejercicio 1, y *grammar.py*, con el código desarrollado para el resto de ejercicios. El fichero *grammar.py* sólo puede ser modificado para añadir los métodos incompletos indicados en los ejercicios 2-5.