Prácticas de Autómatas y Lenguajes. Curso 2021/22

Práctica 2: Análisis Sintáctico

Duración: 6 semanas.

Entrega: Lunes 20 de diciembre

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:

- *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 único objetivo será contar el número de ocurrencias de *números mágicos* en el código (consideraremos número mágico toda constante numérica distinta de 0, 1 y 1j, la unidad imaginaria). Para ello se deberá 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.

Para ello, debemos 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 clase 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.

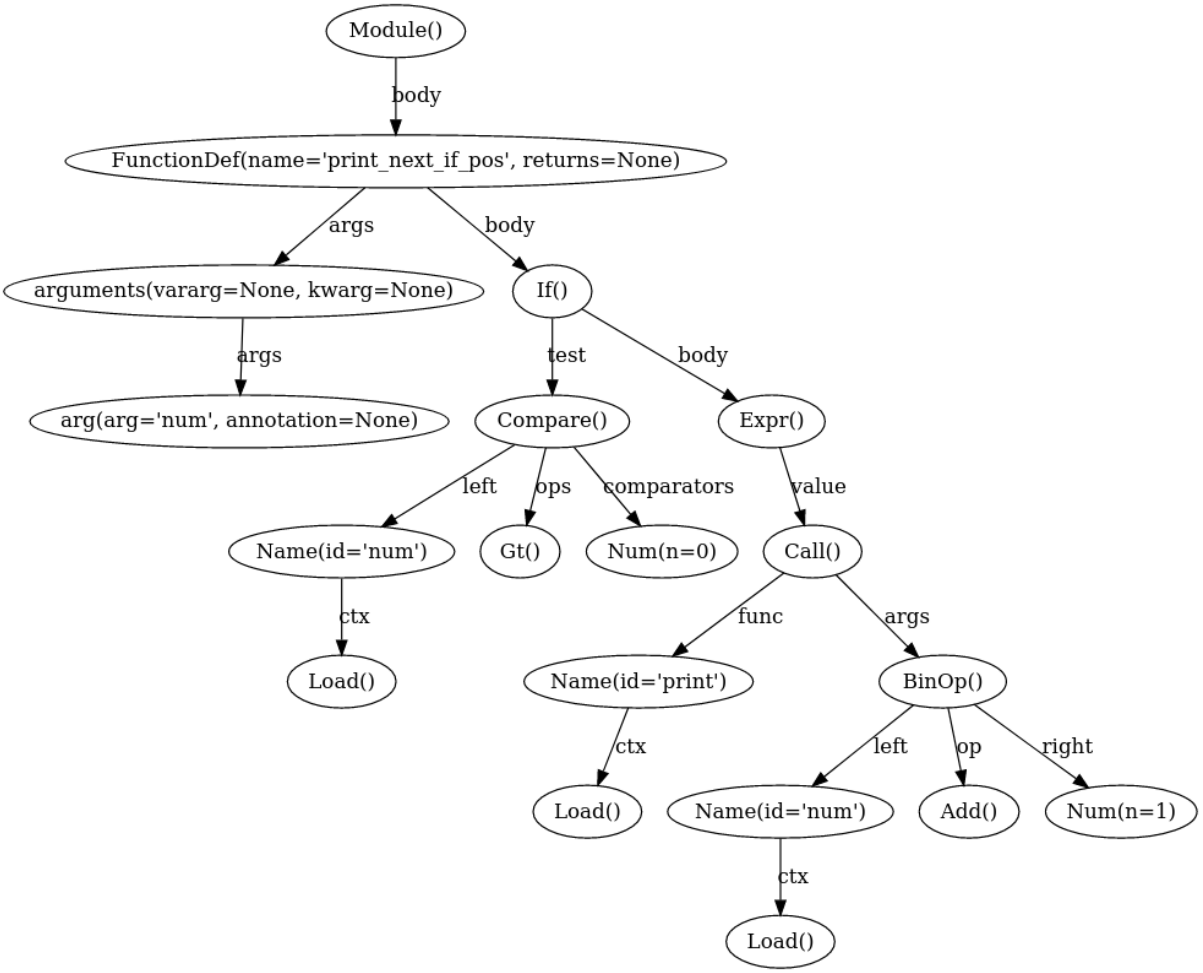
El siguiente código muestra cómo se usaría la clase implementada:

```
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
```

Apartado (b): Usando la misma clase NodeVisitor, cread una subclase “ASTDotVisitor” en el mismo módulo que imprima el AST en el formato dot de Graphviz (os será útil para visualizar los ejercicios siguientes). Por ejemplo, para la siguiente función:

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

debería generar un gráfico como el siguiente:



Como puede verse en la imagen, 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 los 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”. Por ejemplo en el caso anterior el código impreso sería similar al siguiente:

```
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"]
}
```

Apartado (c): Usando la clase NodeTransformer, cread una subclase "ASTReplaceNum" que reciba como argumento un número y reemplace todos los números del código por dicho número. Comprueba el funcionamiento mediante la función "transform_code" que se proporciona:

```
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
```

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

La diferencia entre NodeTransformer y NodeVisitor es que NodeTransformer 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 ilustra el uso de esta clase:

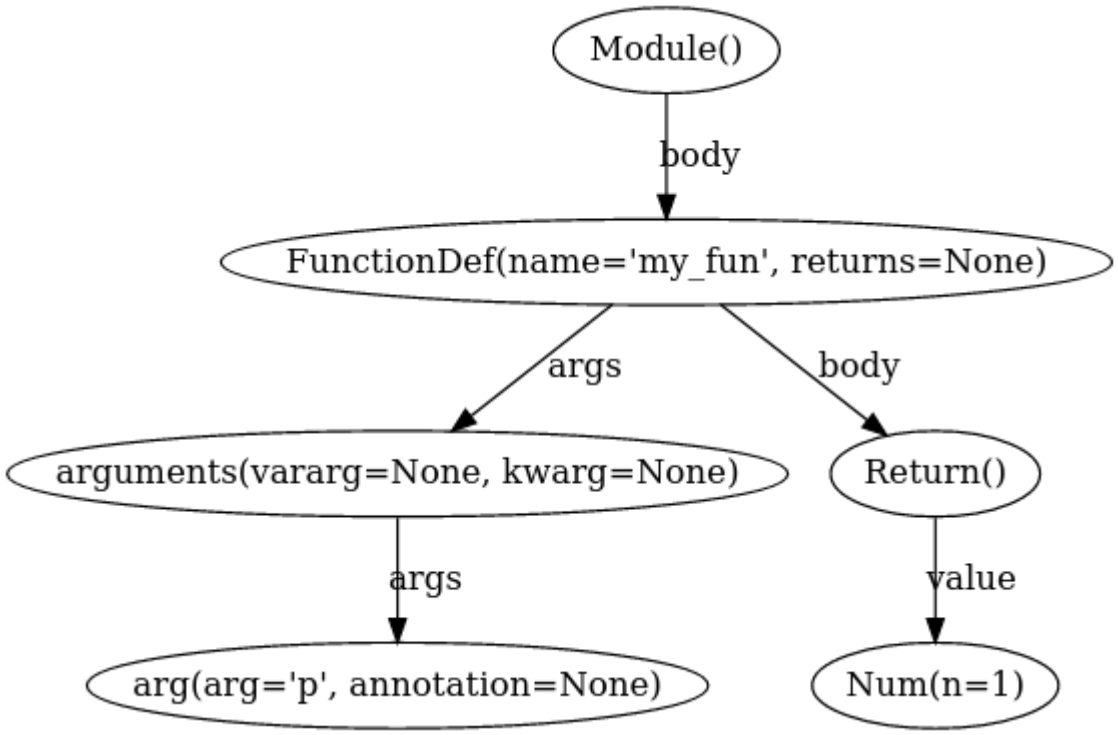
```
import ast
import inspect
from ast_utils import ASTReplaceNum, transform_code

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
```

Apartado (d): Finalmente simularemos la realización de un posible paso de optimización de un compilador. Para ello se debe programar 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.

El siguiente código muestra cómo se usaría esta clase:

```
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)
```

El AST resultante sería:



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.