PROCESADORES DEL LENGUAJE

PRÁCTICA 1 | Introducción al entorno

Reconocedores léxico y sintáctico

Curso 2023-24

Campus de Colmenarejo

uc3m | Universidad Carlos III de Madrid

1/ Introducción

La primera fase de laboratorio de Procesadores de Lenguaje tiene como propósito lograr que el alumno se familiarice con el entorno de desarrollo de la asignatura, además de comenzar con la aplicación de sus conocimientos para el diseño de expresiones regulares y gramáticas básicas de los analizadores léxico y sintáctico con una serie de propuestas de implementación sencillas e introductorias.

2/ Herramientas

Como se introdujo en la primera sesión de laboratorio, los trabajos prácticos de esta asignatura se desarrollarán en el lenguaje Python en combinación con la librería *PLY*[1] que nos ofrece las herramientas *lex* y *yacc*. Estas herramientas son adaptaciones al lenguaje *Python* de las dos herramientas clásicas de construcción de compiladores.

La primera de ellas, *lex*[2], es un generador de analizadores léxicos (o *scanners*), originalmente desarrollada en y para lenguaje C. En muchos aspectos es similar a su homóloga *Flex*[3] con la única particularidad de que esta última es la alternativa en versión de software libre.

Por otro lado, la herramienta *yacc*[4] sirve para generar analizadores sintácticos (o *parsers*) a partir de la especificación de una gramática independiente del contexto. De igual manera que sucede con *lex*, es una implementación de la herramienta original *yacc* y *Bison* (su versión libre), desarrolladas originalmente para el lenguaje C.

Se recomienda en lo sucesivo tener el manual de la librería *PLY*, así como los manuales de las herramientas *Flex* y *Bison*. La mayor ventaja que aporta el uso de estas piezas de software es la comodidad de implementación: únicamente se necesitan dos archivos, que agrupan la especificación de *scanner* y *parser* usando una sintaxis clara y simple.

Originalmente, tanto *lex* como *yacc* se proporcionaban como archivos ejecutables que al ejecutarse sobre los ficheros de especificación de los analizadores generaban el código fuente necesario para construir su implementación. En el caso de *PLY*, se trata de dos módulos de Python que generan directamente los analizadores léxico y sintáctico, y que se integrarán con otros elementos de código *Python* para construir el compilador completo.

---

[1] Creado por David M. Beazley - http://www.dabeaz.com/ply/
[2] *Lexical Analyzer* – Reconocedor léxico
[3] *Fast LEXical analyzer generator*
[4] *Yet Another Compiler Compiler* – Reconocedor sintáctico

3/ Creación del proyecto – *PLY*

A continuación, indicamos los pasos a seguir para crear un proyecto en *Python* en conjunción con la librería anteriormente descrita *PLY*, que nos servirá como base para los reconocedores léxico y sintáctico.

3.1/ Configurando el entorno

Para configurar el entorno de desarrollo será necesario llevar a cabo los siguientes pasos:

I. Instalar un entorno de desarrollo del lenguaje Python, recomendable una versión de Python 3.
II. Existen dos opciones para instalar en el proyecto la librería *PLY:*
    a. Descargar y descomprimir el paquete de *ply-3.11.tar.gz*, copiando la carpeta src/*ply* en nuestro proyecto.
    b. Utilizar algún gestor de librerías de *Python* para instalar el módulo de manera global. Por ejemplo, usando *pip:*

```
pip install ply
```

3.2/ Ejecución del proyecto

Una vez instalado, podemos probar la instalación con los ejemplos que encontramos en *Github* o con alguna de las implementaciones sencilla que han sido realizadas en los laboratorios.

Es importante tener en cuenta que, a pesar de que en alguno de los ejemplos se espera que la entrada de los reconocedores sea introducida por el usuario directamente en la terminal, para la correcta realización de esta práctica deberemos leer la entrada desde un fichero. Para leer el contenido de un fichero, podemos utilizar un código similar al siguiente:

```
import sys

file_name = sys.argv[1]
file = open(file_name)
content = file.read()
```

Para ejecutar el reconocedor léxico, deberemos utilizar un código similar al siguiente:

```
import ply.lex as lex

content = "1 2 3"
lexer = lex.lex()
lexer.input(content)
for token in lexer:
  # token: { name, type, lineno, lexpos }
  print(token)
```

Para ejecutar el reconocedor sintáctico, deberemos utilizar un código similar al siguiente:

```
import ply.yacc as yacc

content = "1 2 3"
parser = yacc.yacc(debug=True)
parser.parse(data)
```

Recordamos que podremos depurar el comportamiento de nuestros reconocedores accediendo a los tokens en el caso del reconocedor léxico, e introduciendo cierta lógica en las funciones de las reglas de producción del reconocedor sintáctico (o utilizar el *flag* de *debug* como se muestra en la imagen anterior lo que generará un fichero *parser.out* con cada ejecución).

4/ Enunciado

En esta sección se explicarán los distintos requisitos de cada uno de los reconocedores, junto con sugerencias para modificaciones más avanzadas. La calificación total de la tarea dependerá de si se cumplen o no los requisitos tanto del reconocedor léxico como del reconocedor sintáctico, mientras que las modificaciones avanzadas serán consideradas como un elemento adicional u opcional.

En esta práctica vamos a construir los reconocedores del formato de texto *AJSON* (*Almost a JavaScript Object Notation*)[5].

```
{
    "this is": "AJSON",
    keys_sometimes_dont_need_quotes: "values always do",
    "you can nest other AJSON": {
        _and_have_other_types_like: 10000,
        or: 12.123,
        "another": 33 <= 0xAA,
        yet_another: NULL
    }
}
```

4.1/ Reconocer léxico

A continuación, se detallan los diferentes *tokens* que ha de reconocer nuestro analizador. Tened en cuenta que es posible utilizar más tokens de los que aquí se detallan para completar la práctica. Es importante revisar la documentación para cumplir todo lo que se especifica a continuación.

4.1.1/ Números

Se reconocerán diferentes tipos de números. Hay que conservar la precisión decimal solo para aquellos valores que así lo requieren.

I. Números enteros: 10, 420, -12, -999, etc.
II. Números reales: 0.1289, .12, -100.001, etc.
III. Notación científica: 10e-1, .1E10, 5E2, 4e-2, etc. (se utiliza “e” o “E”)
IV. Números binarios: 0b101, 0B110110, etc. (comienzan con “0b” o “0B”)
V. Números octales: 0712, 0332, 01121, etc. (comienzan con “0”)

[5] Basado en JSON (*JavaScript Object Notation*)

VI. Números hexadecimales: 0XFED123, 0xAA, 0X1, 0x00F1, etc. (comienzan con "0x" o "0X")

4.1.2/ Cadenas de caracteres

Se reconocerán dos tipos de cadenas de caracteres:

I. Sin comillas
   a. Solo pueden empezar por una letra (mayúscula o minúscula) o una barra-baja (_).
   b. Solo pueden contener letras (mayúsculas o minúsculas), barras-bajas (_) o números.
   c. Solo se pueden usar como claves (más información en el siguiente apartado).

II. Con comillas
   a. Cadenas de cualquier carácter menos el salto de línea y las comillas dobles (**"**). La cadena vacía es válida.
   b. Comienzan y terminan con comillas dobles (**"**).
   a. Se pueden usar como clave o como valor.

4.1.3/ Booleanos

Se utilizarán las palabras reservadas "**TR**" (*True* en *Python*) y "**FL**" (*False* en *Python)*. Se podrán utilizar, de manera completa, tanto en minúscula como en mayúscula.

4.1.4/ Valor nulo

Se utilizará la palabra reservada "**NULL**" (*None* en *Python*). Se podrá utilizar, de manera completa, tanto en minúscula como en mayúscula.

4.1.5/ Operadores de comparación

Todos los operadores de comparación: "**==**" (igual), "**>**" (mayor que), "**>=**" (mayor o igual que), "**<**" (menor que), "**<=**" (menor o igual que).

4.1.6/ Delimitadores

Se utilizarán los siguientes delimitadores: "**{**" (llave de apertura), "**}**" (llave de cierre), "**:**" (dos puntos), "**,**" (coma).

4.2/ Reconocer sintáctico

A continuación, se detalla el formato *AJSON* más en profundidad, el cual será la base de la gramática a desarrollar. También habrá una pequeña inclusión en el analizador sintáctico para incluir operaciones de comparación entre números.

4.2.1/ *Almost a JavaScript Object Notation*

Como comentábamos anteriormente, *AJSON* es un formato de texto que sigue se especifica con las reglas a continuación:

I. Un objeto *AJSON* está delimitado por una llave de apertura y una llave de cierre.
II. El objeto contendrá ninguno o infinitos pares clave/valor
III. Las claves solo puede ser cadenas de caracteres, delimitadas o no por comillas dobles como se explicaba en el apartado del reconocedor léxico.

IV. Los valores podrán ser números, cadenas de caracteres (siempre delimitadas por comillas dobles), booleanos, valores nulos, operaciones de comparación u otros objetos *AJSON* anidados.
V. El par clave/valor se separa con el token dos puntos ":".
VI. Los pares se separan utilizando el token coma ",". Este separador es opcional para los últimos elementos de la serie.
VII. Un fichero *AJSON* puede estar vacío.
VIII. Un objeto *AJSON* puede estar vacío. Traduciremos este caso como un valor nulo.

```
{
  clave_vacia: {},
  clave_numero: 10,
  clave_cadena_caracteres: "string",
  clave_booleano: TR,
  clave_valor_nulo: NULL,
  clave_op_comp: 10 > 11,
  clave_anidada: {
    "clave con comillas": "",
  }
}
```

4.2.2/ Operaciones de comparación

Una operación de comparación consiste en dos números y un operador de comparación. El resultado de esta operación siempre será de tipo booleano.

```
1 > 3               # False
0xFF >= 0b111011    # True
1e-1 == 0.1         # True
077 < 62            # False
```

4.2.3/ Ejecución

Al ejecutar el *parser* ¸se deberá mostrar por consola todos los pares clave/valor. Para los valores anidados, se mostrarán solo los pares finales, siendo su clave la concatenación de las claves anidadas separadas por un punto ".". Para el ejemplo anterior, la salida sería:

```
>> FICHERO AJSON "example.ajson"
{ clave_vacia: None }
{ clave_numero: 10 }
{ clave_cadena-caracteres: string }
{ clave_booleano: True }
{ clave_valor_nulo: None }
{ clave_op_comp: False }
{ clave_anidada.clave con comillas: }
```

Para el caso del fichero vacío se deberá mostrar en la terminal:

```
>> FICHERO AJSON VACÍO "vacio.ajson"
```

4.3/ Modificaciones avanzadas

En esta sección se describen un apartado OPCIONAL que, en caso de ser implementado, podría subir la calificación de la práctica. Cualquier entrega parcial será tenida en cuenta.

4.3.1/ *Arrays* de objetos *AJSON*

Añadir otro posible valor para los pares clave/valor: los *arrays* de objetos. Un array de objetos consiste en una serie de objetos *AJSON*, separados por comas "**,**" y delimitados por los caracteres corchete de apertura "**[**" y corchete de cierre "**]**". A continuación, se detalla un ejemplo:

```
{
  "array": [
    { a: 1, b: 2},
    { a: 1},
  ]
}
```

Y al ejecutarlo, cuál sería la salida en la terminal:

```
>> FICHERO AJSON "arrays.ajson"
{ array.0.a : 1 }
{ array.0.b : 2 }
{ array.1.a : 1 }
```

5/ Entrega

Cada pareja debe entregar todo el contenido de su práctica en un único archivo comprimido, preferiblemente en formato *.zip*

El nombre del comprimido debe seguir el siguiente formato: "**PL_P1_{AP1}_{AP2}.zip**", donde {**AP1**} y {**AP2**} son los primeros apellidos de cada integrante del grupo en orden alfabético.

Al abrir el archivo comprimido, debe verse un único directorio con el mismo nombre que el del archivo. Este directorio debe contener el proyecto completo y funcional, con los archivos de especificación léxica y sintáctica del *scanner* y el *parser*, respectivamente, que satisfagan los requisitos del enunciado.

El proyecto debe poder ejecutarse utilizando el siguiente comando de *Python* en la terminal:

```
python ./PL_P1_{AP1}_{AP2}/main.py {input_file}
```

Se adjuntará una breve (pero completa) memoria, que profundice en el razonamiento y proceso de toma de decisiones de los autores. El formato deberá ser únicamente PDF, y el nombre coincidirá con el del fichero, pero con el sufijo *"_memoria.pdf"*.

Es importante tener en cuenta que los criterios que primarán en la corrección son:

I. Funcionalidad: Todo debe funcionar sin errores y devolver la salida esperada. Se recomienda probar el código con una batería de pruebas, la cual se recomienda entregar en caso de realizarse.
II. Buen uso de las herramientas *PLY*: Demostrar que se conocen y dominan las herramientas trabajadas en clase. Se recomienda revisar los manuales para completar o confirmar el conocimiento impartido en los laboratorios.
III. Exposición justificada: La memoria es un elemento clave de la entrega y deberá reflejar y ayudar al corrector a revisar el trabajo realizado, por lo que se recomienda dedicarle una atención similar que a la propia funcionalidad.

Una vez descomprimido el fichero *.zip*, la estructura de la carpeta debería seguir un formato similar al siguiente (la estructura del proyecto *Python* podría ser diferente, incluyendo más ficheros en caso de considerarlo necesario)

```
/PL_P1_{AP1}_{AP2}
    /test_files
        test1.ajson
        test2.ajson
        ...
    main.py
    ajson_lexer.py
    ajson_parser.py
    PL_P1_{AP1}_{AP2}_memoria.pdf
```

Notas

I. Es muy importante respetar de forma estricta el formato de entrega. De no hacerlo, se corre el riesgo de perder puntos en la calificación final.
II. Cada grupo responderá de forma totalmente responsable del contenido de su práctica. Esto implica que los autores deben conocer en profundidad todo el material que entreguen.
III. Ante dudas de plagio y/o ayuda externa, el corrector podrá convocar al grupo para responder a cuestiones sobre cualquier aspecto de la memoria o código entregados.