PROCESADORES DEL LENGUAJE

PRÁCTICA 1. Introducción al entorno

Reconocedores léxico y sintáctico

Curso 2024-25

Campus de Colmenarejo

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.

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 "scanner"), 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 "parser") 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 del "scanner" y el "parser" usando una sintaxis clara y simple.

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.

CONFIGURACIÓN INICIAL

1. Instalar un entorno de desarrollo del lenguaje Python 3.
2. 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 la raíz de nuestro proyecto.
    b. Utilizar algún gestor de librerías (por ejemplo, pip) de Python para instalar el módulo de manera global.

[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

EJECUCIÓN

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

if len(sys.argv) < 2:
  raise Exception('File name is missing!')

file_name = sys.argv[1]
with open(file_name, mode='r') as file:
  content = file.read()
```

Si queremos mostrar por pantalla los tokens/lexemas reconocidos por el analizador léxico, podemos utilizar un código similar al siguiente:

```
import sys
import ply.lex as lex

# get file content

lexer = lex.lex()
lexer.input(content)

for token in lexer:
  print(token.type, token.value, token.lineno, token.lexpos)
```

Para procesar la entrada con el analizador sintáctico, podemos utilizar un código similar al siguiente. Es importante recordar que debemos construir el analizador léxico en el mismo fichero o bien importarlo desde otro archivo.

Recordamos que se puede depurar el comportamiento utilizando la opción de “debug”, lo que generará un fichero “parser.out” con cada ejecución.

```
import sys
import ply.yacc as yacc

# get file content

# import or build lexer
lexer = lex.lex()

parser = yacc.yacc(debug=True)
parser.parse(content)
```

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 capaces de analizar ficheros con operaciones matemáticas en notación polaca (también conocida como notación de prefijo o notación prefija).

```
'''
This is MyPolisNotationCalculator
'''

+ 3.1 4.5  # This should output 7.6
* / * 0x11 12 4 0x01  # This one combines decimal and hexadecimal, output should be 51

{MEMORY} = 5
{MEMORY} = + / * {MEMORY} 4 10 neg 3
{MEMORY} # Memory is equal to -1
```

RECONEDOR LÉXICO - SCANNER

El analizador léxico deberá ser capaz de:

- Aceptar e ignorar comentarios, de una (#) y múltiples líneas (''')
  - Deben ser ignorados por completo en el análisis léxico.
  - Pueden aparecer a continuación de una expresión válida.

```
# This is one line comment

'''
This is a
multi-line
comment.
'''

+ 3 7  # This is a comment after a valid expression
```

- Reconocer los siguientes tipos de número:
  - ENTEROS: `10` `240` `0` `999` ...
  - REALES: `1.34` `33.0` `0.0023` `100.001` ...
    - Es necesario que el número tenga parte entera y parte decimal.
    - Se utilizará el punto (.) como carácter delimitador.
  - BINARIOS: `0b1011` `0b1` `0b0` `0b00001` ...
    - Comienzan por `0b`

- HEXADECIMALES: `0xFED123` `0xAA` `0x00F1` `0x1011` ...
  - Comienzan por `0x`
  - Las letras que componen el número deben ser mayúsculas.

- Reconocer los siguientes operadores aritméticos: `+` `-` `*` `/`
  - El operador unario de signo negativo (ej. -3) será en este caso `neg`
- Reconocer los siguientes tipos de funciones científicas (operaciones unarias): `exp` `log` `sin` `cos`

```
# Unary operations
log 2
exp 3
neg 0b101
sin0xFF  # There's no need to leave an empty space
cos sin exp 10.5  # We can pass other operations output as the input

'''
Binary operations: Polis notation
With this notation, the operator is the prefix followed by the operands
i.e (+ 3 4) equals (3 + 4) in infix notation
'''

- * 5 3 / 80 10         # This should be equal to 7
* / 10 5 2             # This should be equal to 4
* neg 3 4              # This should be equal to -12
/ * 5 - 4 3 - 8 - 4 1  # This should be equal to 1
+ sin exp 2 3          # This should be close to 3.89385
```

RECONEDOR SINTÁCTICO - PARSER

Se debe diseñar (incluir en la memoria) e implementar, utilizando yacc, una gramática que sea capaz de reconocer expresiones aritméticas en notación polaca, es decir, ficheros con contenido similar a las capturas que se han ido incluyendo en el documento.

Cada expresión estará delimitada por un salto de línea.

Se deberá añadir el código necesario en cada una de las producciones de tal manera que se muestre en la terminal el resultado de las expresiones cuando el analizador realice una reducción.

```
# This file is called "example"
+ + 3 4 5
* 2 / 4 2
sin neg 0xABC

# Empty lines count
+ 1 1

# There can be empty lines without expressions after
```

```
> python main.py example
[Line 2] 12
[Line 3] 4.0
[Line 4] -0.7793149768496849
[Line 7] 2
```

Respecto a las reglas de precedencia, se pide incluir aquellas necesarias para que los resultados de las expresiones sean los esperados. En la memoria se deberá incluir la respuesta detallada a las siguientes preguntas:

- ¿Es necesario el uso de paréntesis en la notación polaca?
- ¿Se necesitan las mismas reglas de precedencia que en la notación infija? ¿Cómo serían para el caso de la notación postfija o polaca inversa?
- ¿Cuál es la notación más sencilla de implementar con una gramática? ¿Y la más compleja?
- ¿Por qué son necesarios los paréntesis en las expresiones de notación infija pero no en la notación polaca?

MODIFICACIONES AVANZADAS

En esta sección se incluyen apartados OPCIONALES que, en caso de ser implementados, podrían subir la calificación de la práctica. Cualquier entrega parcial de cualquiera de los apartados será tenida en cuenta.

- Incorporar el infinito como entrada y posible resultado de las operaciones, utilizando el token "inf" en la entrada (se puede utilizar también en operaciones unarias).
    - Se contemplarán las reglas habituales de operaciones aritméticas con infinito.
    - Se notificará como posible resultado el valor desconocido con el token "nan" en los casos usuales de indefinición con aritmética del infinito. Este token además podrá ser una entrada válida y el resultado será también el valor desconocido.

```
'''
EXTRA 1: Infinity as input or/and output
'''
+ inf 4          # equal to inf
/ inf neg 1      # equal to -inf
/ 12 0           # equal to inf
/ 10 inf         # equal to 0
* inf neg inf    # equal to -inf

''' UNKNOWN (nan) examples '''
# All these expressions are equal to nan
- inf inf
+ neg inf inf
/ 0 0
* 0 inf
/ inf inf
* 0 nan
* inf nan
```

- Incorporar una variable de memoria – {MEMORY} – de manera que los operadores participantes en las operaciones podrán ser los números literales y dicha variable de memoria, cuyo valor puede ser modificado a lo largo del programa.
    - Existirán pues dos tipos de sentencia, expresiones matemáticas y asignaciones.
    - Las asignaciones comienzan por el nombre de la variable – {MEMORY} -seguido del signo igual y terminando con una expresión matemática.
    - El valor por defecto de esta variable es 0, por lo que puede ser usada en una expresión sin haber aparecido con anterioridad en una asignación.

```
{MEMORY} = 5                                # Memory is equal to 5
{MEMORY} = * neg + * 7 {MEMORY} neg 5 neg 1   # Memory is equal to 30
/ {MEMORY} 2                                # Should print 15
```

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 `AP1_AP2_PL_P1.zip`

- AP1 y AP2 son los primeros apellidos de los integrantes en orden alfabético.
- Dentro del fichero zip, habrá una única carpeta con el mismo nombre.

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, una vez descomprimido el entregable:

```
> python ./{AP1}_{AP2}_PL_P1/main.py {input_file}
...
```

Se adjuntará una breve y directa (pero completa) memoria, que profundice en el razonamiento y proceso de toma de decisiones de los autores. Solo se aceptarán memorias en formato PDF, y el nombre del documento 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. Si no es posible ejecutar el código tal y como se indica en la imagen anterior, no se corregirá la parte práctica y solo se tendrá en cuenta, en cualquier caso, la memoria.

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.

    a. Es importante recordar que en el enunciado se pide expresamente incluir ciertos apartados en la memoria en las secciones correspondientes.

    b. Las partes opcionales, más aún cuando se entreguen incompletas, requerirán siempre una justificación debida de lo que se ha intentado hacer, cómo y por qué, así como qué materiales se han utilizado para implementar partes más avanzadas aun no vistas en clase.

IV. Batería de pruebas: Se deben incluir una serie de ficheros de prueba que demuestren que los autores han comprobado que su implementación reconoce y procesa correctamente las entradas esperadas, así como rechaza y muestra los errores correspondientes en el caso contrario.

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)

```
/{AP1}_{AP2}_PL_P1
    /test_files
        test1
        test2
        ...
    {AP1}_{AP2}_PL_P1_memoria.pdf
    main.py
    lexer.py
    parser.py
```

Es muy importante respetar de forma estricta el formato de entrega. De no hacerlo, se corre el riesgo de perder puntos de calificación.

Cada grupo responderá de forma totalmente responsable del contenido de su práctica. Esto implica que los autores deben conocer en profundidad todo el material creado por ellos.

Ante dudas de plagio o ayuda externa, el corrector podrá convocar al grupo para responder a cuestiones sobre cualquier aspecto de la memoria o código entregados.