Universidad
Carlos III de Madrid
www.uc3m.es

PROCESADORES DEL LENGUAJE

PRÁCTICA 1. Introducción al entorno
Reconocedores léxico y sintáctico

Olivia Grima Pérez - 100474858

Íñigo García-Velasco González - 100461095

Grupo 80

ÍNDICE

Introducción

Esta primera práctica nos ha introducido al entorno de desarrollo de la asignatura. Nos hemos familiarizado con la librería *PLY* de Python, trabajando con sus distintas funcionalidades. Hemos construido un reconocedor léxico (scanner), capaz de reconocer correctamente los distintos léxicos e ignorar aquellos que no tienen que ser reconocidos, y sintáctico (parser), capaz de aplicar las distintas reglas sintácticas para procesar expresiones matemáticas en notación polaca.

En esta memoria se detalla el proceso llevado a cabo para construir el parser y scanner, justificando adecuadamente las decisiones tomadas y su correcto funcionamiento mediante casos de prueba, garantizando que las expresiones son analizadas de manera adecuada, generando los resultados esperados.

Reconocedor léxico - scanner

Con el fin de llevar a cabo la calculadora polaca expuesta en el enunciado, lo primero que hicimos fue crear el constructor de la clase *lexer* en el que encontramos la función de PLY lex.lex para crear el lexer y fuera del *__init__* creamos los tokens necesarios para que nuestro lexer pueda reconocer tanto los números y funciones científicas como los operadores que recibimos como entrada, así como valores especiales, que son los que incluimos posteriormente cuando realizamos las modificaciones avanzadas.

Una vez definidos los tokens, lo siguiente era crear la expresión regular que corresponde a cada uno de ellos, esto lo hicimos con una asignación normal para los operadores aritméticos y el signo de igual (el cual nos servirá para el *memory*), y en forma de función para los tipos de números y las distintas funciones científicas.

Lo siguiente que hicimos fue todo el manejo de los saltos de línea, para el cual utilizamos la función *t_NEWLINE* que nos permite llevar la cuenta de en qué línea se sitúa el token que se está leyendo para facilitar así el manejo de errores.

Se nos pedía que el analizador léxico fuese capaz de aceptar e ignorar comentarios de una y múltiples líneas, para lo cual tuvimos que implementar varias funciones.

La función que utilizamos para ignorar los comentarios de una sola línea es muy básica, simplemente hacemos que ignore todo lo que comience con un "#". Sin embargo, la lógica detrás de los comentarios multilínea es algo más compleja, ya que para desarrollar de manera correcta hemos necesitado una previa declaración de estados y 3 funciones.

La declaración de estados nos sirve para el manejo de comentarios multilínea, ya que, nos permite definir reglas concretas para ignorar comentarios dentro del estado "comment" y con el estado "exclusive" le indicamos al lexer que solo va a estar en este estado de manera exclusiva, cuando detecte las 3 comillas.

Partiendo de esa base, la primera función que utilizamos para poder ignorar los comentarios multilínea es *t_MULTI_LINE_COMMENT* que lo que hace es detectar las 3 comillas y modificar el estado a comment, para así identificar que estamos entrando en un comentario multilínea y poder detectar todo el comentario como tal.

La siguiente función que utilizamos es la función *t_comment_MULTI_LINE_COMMENT*, con la que identificamos el final del comentario multilínea, gracias a las otras 3 comillas, y hacemos que el lexer vuelva a su estado inicial.

Por último tenemos la función con la que creamos el manejo del contenido de los comentarios, *t_comment_CONTENT*. Aquí lo que hacemos es utilizar una expresión regular que nos permite detectar texto de todo tipo y saltos de línea hasta llegar a las comillas finales.

Aparte de esto, cuando nos encontramos dentro del estado "comment" tenemos un *t_comment_ignore* para ignorar los espacios y tabulaciones que aparecen, así como una función que se encarga del manejo de errores léxicos dentro de este estado.

De la mano con lo que acabamos de mencionar, tenemos una funcionalidad igual para cuando estamos fuera del estado de comentarios. Tenemos un *t_ignore* que actúa exactamente igual que el previamente mencionado y una función de error, que se activa en caso de que el lexer analice algo que no coincide con los tokens indicados y nos devuelve la línea en la que se encuentra el error.

Para finalizar con el lexer, lo último que implementamos fue una función de *test* para comprobar que los tokens se estaban recibiendo correctamente y tras hacer unas cuantas pruebas y ver que todo funcionaba tal y como se indicaba en el enunciado decidimos continuar construyendo el parser.

Reconocedor sintáctico - parser

Para poder procesar las expresiones matemáticas en notación polaca, tuvimos que diseñar e implementar una gramática mediante un *parser*, utilizando *yacc,* que fuera capaz de reconocer las expresiones en notación polaca y calcular su resultado esperado.

Lo primero que hicimos fue diseñar dicha gramática para luego poder implementarla en código. La gramática creada fue:

G = ({+,-,*,/, "neg", "exp", "log", "cos", "sin", =, MEMORY, entero, real, binario, hexadecimal, inf, nan}, {lista_expresiones, expresion, operacion_binaria, operacion_unaria, asignacion}, programa, P)

P = { **programa ::=** lista_expresiones | λ

lista_expresiones ::= expresion lista_expresiones | expresion

expresion ::= operacion_binaria | operacion_unaria | numero | variable | asignacion

operacion_binaria ::= operador_bin expresion expresion

operacion_unaria ::= operador_un expresion

asignación ::= variable símbolo expresion

numero ::= entero | real | binario | hexadecimal | inf | nan

operador_bin ::= + | - | * | /

operador_un ::= neg | exp | log | sin | cos

símbolo ::= =

variable ::= MEMORY

}

Una vez hecho esto, y con nuestro *lexer* ya hecho, empezamos a programar el *parser*. La gramática definida nos permite calcular expresiones aritméticas utilizando los operadores comunes (+,-,/,*), funciones científicas (cos, sin, exp, log) además del operador unario de signo negativo (neg). Estas operaciones se procesan de acuerdo con las reglas sintácticas definidas.

Para empezar, importamos la lista de tokens definida en la clase *lexer* y creamos el constructor donde creamos una instancia de la clase *lexer* y construimos el *parser* con *yacc*.

Seguidamente, empezamos con las reglas sintácticas. Primero tenemos una regla inicial, que sirve como punto de entrada del *parser* en la que definimos que el programa está compuesto por una lista de expresiones. También incluimos que el programa puede ser un conjunto vacío, por lo que añadimos una regla *p_empty* para que reconozca el conjunto vacío y no de un error de sintaxis de entrada en el *parser.* Luego definimos otra regla *p_lista_expresiones* que nos permite evaluar múltiples expresiones en una misma ejecución.

Una vez definidas estas reglas, pasamos a las reglas de las expresiones. Primero de todo tenemos la regla *p_exp_binaria* para las expresiones binarias. Aquí definimos que una expresión tiene que contener un operador del tipo +,-,/,* seguido de dos operandos. En esta regla hacemos las operaciones correspondientes según el operando, utilizando la librería *math*. Es decir, si el operando es "+" sumamos el segundo y tercer elemento ya que son los que corresponden con los operando según la notación polaca. Esto lo hacemos para cada operando y también tenemos en cuenta si es una operación que se puede hacer o no, como por ejemplo dividir por cero, en cuyo caso devuelve "nan". También, aunque se explicará más en detalle en el apartado de modificaciones avanzadas, se contempla el uso del infinito y nan. La siguiente regla es *p_exp_unaria* en la que definimos las operaciones unarias con neg, log, exp, cos y sin y que solo necesitan un operando. Después seguimos el mismo procedimiento que en la regla anterior, haciendo las operaciones pertinentes utilizando *math.* La próxima regla es *p_exp_número* donde definimos los distintos números que nos pueden entrar (entero, real, binario, hexadecimal, infinito y nan). Estos son los únicos tipos de operandos que

podemos tener, por lo que si en vez de meter un número de los expresados, como por ejemplo una letra, ni el lexer ni el parser lo reconoceran. Luego tenemos dos reglas más para tener en cuenta la variable *memory*, aunque estas reglas se van a explicar en otro apartado.

Por último, tenemos la regla para el manejo de errores, *p_error,* en el que si se encuentra un error sintáctica, lanzará un mensaje de error del *parser* y una regla de prueba *p_test* para comprobar que el parser funciona correctamente e imprime los resultados de las expresiones con su correspondiente número de línea.

¿Es necesario el uso de paréntesis en la notación polaca?

No, precisamente el uso de la notación polaca evita utilizar paréntesis. En la notación polaca el orden ya está implícito, por lo que los paréntesis son innecesarios. Esto es porque en la notación polaca el operador precede a sus operandos entonces si tienes una operación con tres número y dos operandos como por ejemplo, + * 3 4 5, la multiplicación se va a hacer con el 3 y 4 ya que son los operandos que le preceden y por lo tanto la suma se va a hacer entre el 5 y el resultado de la operación anterior.

¿Se necesitan las mismas reglas de precedencia que en la notación infija? ¿Cómo serían para el caso de la notación postfija o polaca inversa?

No, puesto que la notación polaca ya está, implícitamente, definiendo el orden de precedencia al escribir la expresión. Como he explicado en la pregunta anterior, cada operador necesita dos operandos y esos operandos son los que le preceden directamente. En la notación infija si se necesitan esas reglas ya que los operadores están entre los números por lo que necesitas las reglas para saber qué operaciones hacer primero, porque un número puede estar entre medias de dos operadores y por tanto no saber si tiene que hacer la operación de la derecha o la de la izquierda. En el caso de la notación polaca inversa, sería igual que en la notación polaca, no se necesitan reglas de precedencia ya que el orden se establece al escribir la expresión.

¿Cuál es la notación más sencilla de implementar con una gramática? ¿Y la más compleja?

La notación más sencilla de implementar con una gramática sería polaca inversa puesto que es la más intuitiva ya que primero se apilan los operandos y luego se multiplica por el operador que precede. No haría falta usar paréntesis ni reglas de precedencia. La notación más compleja para implementar con una gramática es la notación infija ya que esta si que necesita reglas de precedencia y paréntesis porque los operadores están entre medias de los operandos, entonces necesitamos las reglas o los paréntesis para saber qué operación hacer primero.

¿Por qué son necesarios los paréntesis en las expresiones de notación infija pero no en la notación polaca?

Los paréntesis en las expresiones de notación infija son necesarios para saber que operaciones tenemos que hacer primero. Es decir, en la notación infija, podemos tener el caso en el que tengamos tres operandos con dos operadores en medio de cada operando. Sin los

paréntesis no sabríamos qué operación realizar con el operando del medio. En la notación polaca, no haría falta los paréntesis porque el orden en el que escribas los operadores y operandos define qué operaciones hay que hacer primero.

Modificaciones avanzadas

En el enunciado de la práctica se exponían dos modificaciones avanzadas, las cuales eran opcionales pero que nosotros nos hemos lanzado a hacer.

Infinito

Una de las modificaciones avanzadas que se indicaba en la práctica era la incorporación del infinito como entrada y posible resultado de las operaciones, utilizando el token "inf", y cómo "nan" es un posible resultado cuando operas con infinitos, nos pedían también incorporarlo como token.

Lo primero que hicimos fue añadir esos dos tokens en el *lexer* creando sus expresiones regulares. Luego tuvimos que contemplar todas las reglas de las operaciones con infinitos, viendo cuando el resultado es "nan" y cuando son el propio infinito. Una vez hecho esto, tuvimos que añadir estas reglas a las reglas sintácticas del *parser*.

Primero, añadimos los dos tokens a la regla *p_exp_numero* para que el *parser* los reconozca como posibles entradas. Luego, hicimos posible que el infinito se convirtiera en negativo utilizando el token "neg". Para ello, en la regla de expresiones unarias indicamos que si el token que sigue a un "neg" es un "inf", cambiamos el token a "-inf", para así cuando hagas las operaciones binarias, podamos distinguir entre infinito y menos infinito.

Posteriormente, añadimos a la regla *p_exp_binaria* todas las posibles operaciones con "inf" ,"-inf" y "nan". Por ello, para cada operador (+,-,/,*), evaluamos las distintas maneras en las que se puede operar con ellos, entre ellos y con números normales para así poder devolver los resultados congruentes y poder operar con estos tokens. Hemos hecho lo mismo para las operaciones unarias del tipo log, cos y sin.

Memory

La otra modificación que nos pedían implementar era el uso de una variable que guardase el resultado de la operación anterior y poder usarla como operador participante en las expresiones. Básicamente, esta variable tiene dos tipos de sentencia, expresiones matemáticas y asignaciones. Para las asociaciones tienen que tener el nombre de la variable, {MEMORY}, seguido del signo igual, =, y terminado con una expresión matemática, de las que hemos definido en el parser, ya sea

solamente un número, una operación o un conjunto de operaciones. El otro tipo de sentencia es utilizarla en una expresión como un número más.

Para incorporar esta variable, primero la hemos tenido que añadir tanto la variable como el signo del igual como un token más, creando su expresión regular correspondiente en el *lexer*. Luego, en el *parser* hemos incorporado dos reglas necesarias para asegurar el correcto funcionamiento de la variable, con sus dos tipos de sentencias y hemos inicializado una variable en el __*init*__ para poder almacenar los valores y . La primera regla es *p_exp_variable* la cual permite recuperar el valor de la variable *{memory}* y la segunda regla es *p_exp_memory* la cual nos permite asignar un valor a la variable para poder usarla posteriormente. Esta segunda regla requiere que la variable memory tenga después un "=" y luego una expresión para poder asignar a la variable un valor. En este caso, asignamos a la variable local que hemos creado en el __*init*__ el valor de la *expresión* que puede ser desde un número a un resultado de una operación.

Casos de prueba

Para probar que el código implementado realiza las funciones requeridas, hemos realizado unos casos de prueba bastante completos con los que hemos podido comprobar que obtenemos los resultados esperados tras mandar los archivos y ejecutarlos.

Para probar cada funcionalidad de forma clara, hemos creado 4 ficheros de casos de prueba.

El primero es test1_comentarios.txt, en el que lo que probamos más concretamente es que el manejo de los comentarios sea correcto y se ignoren todos. Comprobamos que en efecto, en la salida no se ven los comentarios, solo los resultados de las operaciones con su respectiva línea.

El segundo fichero que hemos creado es test2_basico.txt, en el que hemos incluido casos de prueba para la parte "obligatoria" de la práctica, como los tipos de números, las operaciones unarias y las binarias (que son las operaciones que conocemos con dos operandos o más y sus respectivos operadores), sus combinaciones entre sí incluyendo los casos básicos en los que la salida debería ser nan.

El siguiente fichero que tenemos es el test3_infinito.txt. En este fichero hemos implementado los tests necesarios para comprobar que las operaciones que incluyen infinito (y nan) funcionan. Lo hemos estructurado de tal forma que lo dividimos con títulos para especificar qué casos probamos (sumas, restas, etc) y dentro de estos, agrupamos por salidas (inf, -inf, nan o 0).

Nuestro cuarto fichero de pruebas es test4_memory.txt, en el que hacemos varias comprobaciones del Memory, primero probando que valga 0 cuando todavía no se le ha asignado ningún valor y luego viendo que guarde bien los valores.

Por último, tenemos 4 tests en los que comprobamos diferentes errores de salida. El primero test5_errores1.txt, contiene errores únicamente de lexer, ya que esos tokens no están definidos, tras esto debería aparecer un error por pantalla de que el parser no ha podido procesar la entrada debido a que no le hemos enviado nada.

El segundo, test6_errores2.txt, contiene un error de lexer que es por la palabra “hola” seguido de uno de sintaxis. El error de sintaxis sucede porque en el fichero tenemos un 2 + 2, que debería dar error porque el lexer reconoce los tokens pero como esa estructura no está incluida en la gramática no se puede procesar, después de dar syntax error deberíamos ver por pantalla un error de entrada ya que el parser no ha procesado nada.

El tercer fichero de errores, test7_errores7.txt, contiene un + 2 € 2, lo que nos debería devolver primero un lexer error al no reconocer el token “€” y después debería imprimir el resultado 4, ya que según como tenemos el código del error realizado, el carácter que el lexer no identifica como token se ignora, por lo que se le envía al parser una gramática correcta que puede leer.

El último fichero con el que comprobamos un error de salida es el test8_vacio.txt, este es simplemente un test en el que no hacemos nada, para ver si el parser indica que no se puede procesar la entrada.

Conclusión

Esta primera práctica nos ha servido para familiarizarnos con los conceptos básicos de la asignatura.

Es cierto que no ha sido una práctica excesivamente complicada, pero sí que hemos tenido algún que otro problema al intentar implementar algunas cosas, como por ejemplo, al hacer que se ignorasen los comentarios multilínea, lo que finalmente conseguimos con los estados; y para imprimir el resultado con el número de línea, ya que, tras tener todo el parser hecho, tuvimos que cambiar la forma de devolver los resultados, para lo que tuvimos que modificar gran parte del código.

En un inicio pensábamos que nos iba a costar bastante más crear las modificaciones avanzadas, pero finalmente no nos supuso tanto esfuerzo, simplemente tuvimos que ir caso a caso viendo qué resultados se obtendrían y añadiendo las condiciones para ello.

En conclusión, ha sido una práctica con la que hemos podido aprender cómo funcionan los analizadores léxico y sintáctico para poder crear programas más complejos en un futuro utilizandolos.