uc3m | Universidad Carlos III de Madrid

PROCESADORES DEL LENGUAJE

Práctica 1

Grupo 80

ID Equipo:

Fecha de entrega:

Pedro Hernandez Bernaldo (100432150@alumnos.uc3m.es)
Ignacio Marín de la Bárcena Pérez (100432039@alumnos.uc3m.es)

Índice

Introducción

El objetivo de esta primera práctica es familiarizarnos con las herramientas *PLY lex - yacc*, cuyo uso va enfocado a la parte práctica de la asignatura, y en específico, el desarrollo de la primera práctica. Estas herramientas son adaptaciones a Python de herramientas clásicas de construcción de compiladores. *Lex* es un generador de analizadores léxicos que originalmente fue desarrollada en C, y *yacc* sirve para generar analizadores sintácticos a partir de la especificación de una gramática independiente del contexto

Respecto a la práctica en sí, desarrollaremos una calculadora basada en las herramientas previamente nombradas, constituida por un analizador léxico y sintáctico, capaz de leer la entrada con expresiones numéricas y calcular el resultado de las operaciones aritméticas habituales. La entrada para poder probar los resultados de todas las operaciones es el archivo *input.txt*, en el que hemos incluido todas las operaciones. Para poder probar el correcto resultado de estas, corra el código del fichero *calc.py.*

Reconocedor léxico

En la sección del reconocedor léxico hemos de desarrollar 2 funciones principales:

a) Aceptar e ignorar comentarios: Estos empiezan por el carácter **'%'** y se extienden hasta el final de la línea.

b) Incorporar diferentes sistemas numéricos: octal, hexadecimal, binario y números reales. Cada uno de los sistemas se definirá de forma diferente y deben de ser capaces de realizar las operaciones aritméticas básicas sin problema.

Aceptar e ignorar comentarios

Para tratar e ignorar los comentarios hicimos uso de la regla especial de token ***t_ignore.*** Regla a la cual denominamos ***t_ignore_comment*** que nos permite ignorar todas las líneas que presenten un comentario. Su funcionamiento consiste en reconocer el símbolo **'%'** para posteriormente ignorar todos aquellos caracteres que se sitúen detrás del carácter, ya que se trata de un comentario que el reconocedor léxico no ha de tener en cuenta.

Para comprobar el correcto funcionamiento realizamos diferentes pruebas por medio de incorporar diferentes comentarios en el archivo de entrada ***"input.txt"***. Dichas pruebas consisten en probar todos aquellos casos en el que los comentarios se puedan encontrar obteniendo como resultado que se ignoren todos los comentarios no produciendo así un error.

Incorporar sistemas numéricos

Para el reconocimiento de los diferentes sistemas numéricos, modificamos la función ***t_NUMBER(t)***. Función que se encarga de leer los posibles tokens de entrada referentes a los números y de comprobar de qué tipo se tratan, para posteriormente transformarlos y poder realizar las operaciones. La expresión regular que empleamos es la siguiente:

$$r'0[0-7]+|0[xX][\da-fA-F]+|0[bB][01]+|\d+(\.\d+)?([eE][+-]?\d+)?'$$

Se encarga de reconocer los diferentes formatos de sistemas numéricos (octal, hexadecimal, binario y real). Cada elemento separado por el carácter **"|"** representa un patrón de reconocimiento diferente:

- **0[0-7]+**: número octal que comienza con un cero y seguido por uno o más dígitos del 0 al 7.
- **0[xX][\da-fA-F]+**: número hexadecimal que comienza con "0x" o "0X" seguido por uno o más dígitos hexadecimales (del 0 al 9 y de la a a la f o A a F).
- **0[bB][01]+**: un número binario que comienza con "0b" o "0B" seguido por uno o más dígitos binarios (0 o 1).
- **\\d+(\\.\\d+)?**: un número decimal que consta de uno o más dígitos seguido de un punto decimal y uno o más dígitos decimales opcionales.
- **([eE][+-]?\d+)?**: una notación científica opcional que consta de una letra "e" o "E", seguida opcionalmente por un signo más o menos y uno o más dígitos.

Una vez reconocida la sentencia de entrada, la lógica que sigue la función para leer y transformar los números diferentes sistemas numéricos es por medio de un condicional ***"if"***, que al cumplirse transforma el tipo del token al necesario y en la base correspondiente.
Un ejemplo en pseudocódigo aplicado sería el siguiente:

```
elif t.value.startswith(('0B','0b')):
        t.value = int(t.value,2)
```

Código en el que Comprobamos si el valor del token actual comienza con **'0B'** o **'0b'** (que son prefijos comunes para indicar un número binario). Si la condición es verdadera, convierte el valor del token en un número entero utilizando la función ***int()*** y especificando la base de la conversión (en este caso, la base 2, que corresponde al sistema binario). Luego, el valor del token se actualiza con este número entero resultante, y así con el resto de sistemas numéricos.

Reconocedor sintáctico

En la sección del reconocedor sintáctico hemos de desarrollar 3 funciones principales:

a) Lectura de archivos completos incluyendo saltos de línea: modificaremos la gramática añadiendo los token y reglas de producción necesarias.
b) Añadiremos funciones científicas básicas: implementar en nuestra calculadora funciones como el seno, coseno, logarítmo o exponencial, mediante la incorporación de nuevos símbolos y acciones a la gramática.
c) Eliminación de reglas de precedencia: modificaremos la gramática para evitar la necesidad de utilizar explícitamente las reglas de precedencia para construir la calculadora.

Lectura de archivos incluyendo saltos de línea

Lo primero que realizamos fue pasar de leer el archivo de entrada ***"input.txt"*** línea por línea al archivo completo usando la instrucción ***"fread()"***. Una vez realizado dicho cambio, creamos el token ***"SALTO_LINEA"***, el cual se reconoce cada vez que se recibimos un salto de línea **"\n"** por entrada. Finalmente, modificamos la gramática y creamos nuevos statements para poder reconocer los diferentes casos en los que se puede producir un salto de línea. Creamos dos reglas. La primera se basaba en que un archivo de texto se puede componer de varias sentencias separadas por un salto de línea (esto lo logramos mediante recursividad a derechas ya que es la permitida por el analizador **"LARL"** construido). Mientras que la segunda regla, permite reconocer que el documento de entrada empiece con un salto de línea. Por último, y una vez creadas ambas funciones realizamos diferentes pruebas para comprobar que no saltara ningún error y todo funcionaba correctamente.

Las reglas que añadimos a la gramática fueron las siguientes:

```
'statements : statement SALTO_LINEA statements

            | statement'

'statement : \n'
```

Funciones científicas básicas

Llegados a este punto, se nos pedía añadir funciones científicas básicas a la gramática, Más concretamente funciones exponenciales, logaritmos y trigonométricas (seno y coseno). Para ello, creamos 4 nuevos tokens a los cuales denominamos **"EXPONENCIAL"**, **"LOGARITMICA"**,**"SENO"** y **"COSENO"**. Los cuales permiten reconocer este tipo de funciones cuando se pasan sus símbolos correspondientes (***'exp'***,***'log'***,***'sin'*** y ***'cos'***). Una vez ya definimos como iban a ser los tokens y como iban a ser las entradas que recibiría el analizador, creamos una regla para reconocer este tipo de sentencias. Para ello, declaramos la función de la regla y los casos en los que se iba aplicar, diferenciándolos por medio del uso de condicionales para separar entre los 4 casos posibles. Para terminar, nos vimos en la

necesidad de importar la librería ***"math"***, para que el analizador realizase las operaciones y poder comprobar que las diferentes pruebas realizadas nos dieran los resultados esperados.

El pseudocódigo aplicado es el siguiente:

```
import math

def p_nombre_función(p):
     'operando | EXPONENCIAL "(" expression ")"
               | LOGARITMICA ...  '
     if p[1] == 'exp':
          p[0] = math.exp(float(p[3]))
     elif ...
```

Finalmente aclarar que las funciones trigonométricas implementadas son en radianes, así como la función logarítmica en base 10.

Eliminación de reglas de precedencia

Para eliminar las reglas de precedencia y que el analizador siga funcionando de forma correcta y proporcionando la salida correcta modificamos las reglas existentes y creamos 2 nuevos símbolos no terminales (***"operando"*** y ***"multiplicador"***). Lo primero fue crear la precedencia dentro de los símbolos terminales:

```
expression : multiplicador
multiplicador : operando
```

Una vez creada dicha precedencia procedemos a modificar las existentes y crear una nueva regla en la que definimos el signo de los operandos (***positivo o negativos***). En primer lugar, dividimos la función de la expresión original en sumas y restas por un lado, y multiplicación y división por otro. Para finalmente, sustituir las reglas añadiendo que el nuevo símbolo no terminal operando genere los diferentes tipos de números así como las funciones.

Una vez realizadas todas las transformaciones a la gramática, procedemos a eliminar las precedencias marcadas por el nombre **"precedence"** y a ejecutar las diferentes pruebas realizadas a lo largo del proyecto con el objetivo de obtener como resultados esperados aquellos resultados previamente obtenidos y ningún error. Lo que resultó en que las pruebas fuesen satisfechas al no existir ningún conflicto entre las reglas y poderse generar las tablas "**LARL"** como el **"parser"**.

Modificaciones avanzadas

En esta última sección de la práctica hemos de desarrollar 2 funciones principales:

a) Incorporación del infinito: como entrada y posible resultado de las operaciones, utilizando el token *INF* (+-). Además deberemos incorporar el token *UNK*, que hace referencia a indeterminaciones como *0/0* o *INF - INF*.
b) Incorporación de memoria: como una variable a la que asignamos un valor y almacena el resultado de las operaciones que realicemos, si así lo deseamos.

Incorporación del infinito y desconocido

Una vez ya generadas las posibles operaciones y funciones de la calculadora, procedemos a incorporar los parámetros infinito (**"INF"**) y desconocido (**"UNK"**) para poder realizar operaciones con los mismos. En primer lugar, generamos ambos tokens y las funciones del analizador que leerían la posible entrada de dichos tokens.

Posteriormente, modificamos dos funciones (reglas) del analizador, `p_expression_binop`, y `p_multiplicador_operando`. Con el objetivo que satisficiera todas las posibles combinaciones en las que se pudiera dar una entrada con el valor infinito o desconocido. Siendo necesario realizar diferentes pruebas para que se solucionaran todas las indeterminaciones posibles que se pasar como entrada. A su vez, al realizar dicha comprobación para el correcto funcionamiento de ambos posibles valores, tuvimos que modificar el tipo de algunas de las variables a lo largo analizador y forzandolos a ser tipo **"float"** para que se pudieran realizar las operaciones deseadas. El método aplicado para la resolución de los posibles casos en los que se podría recibir algunos de estos valores y reconocer la sentencia, fue por medio del uso de condicionales. Un ejemplo en pseudocódigo sería:

```
if p[2] == '+':

    if [p1] == 'INF' or p[3]:

        p[0] = 'INF'

    elif p[1] == -float('INF') and ...
```

Incorporación de una variable memoria

El último paso para completar el proyecto y por ende la calculadora, consistía en incorporar una variable memoría y que esta se pudiera usar en las diferentes operaciones que

se pueden dar en el archivo de entrada. Para ello, creamos el token **“MEMORIA”** y la función que lo define para poder leerlo. Una vez creado el token, reservamos la variable **‘MEM’** en el diccionario de nombres inicializado la variable a **‘0’**. Posteriormente, modificamos dos de las funciones ya predefinidas con la intención de poder incluir esta nueva variable a las operaciones. Cambiando las reglas de las funciones `p_statement_asign` por `'statement : MEMORIA "=" expression'` y la función `p_operando_name(p)` en la que modificamos la regla por `'operando : MEMORIA'`. La realización de ambos cambios permitió tanto asignar valores a la variable **“MEM”** como usarla para realizar operaciones. Su correcto funcionamiento se comprobó mediante el uso de diferentes pruebas en las cuales se obtuvo siempre el resultado obtenido.

Conclusiones y problemas encontrados

Este proyecto nos ha permitido no solo conocer y entender las herramientas de trabajo ***parser*** y ***yacc*** en el lenguaje de programación python. A su vez, nos ha permitido entender más el correcto funcionamiento de un analizador LALR así como de los reconocedores sintácticos y léxicos a la par que nos ofrecía poner en práctica lo aprendido a lo largo de las clases prácticas y teórica en un ámbito más laboral y que sirva de cara a nuestro futuro más próximo.

Por otro lado, a lo largo del desarrollo de este proyecto nos enfrentamos a diversos problemas. El primero fue entender el funcionamiento de los archivos y del analizador en sí, pero una vez entendido el correcto funcionamiento del código, la práctica se avanzó sin ninguna dificultad mayor. Y en segundo lugar, realizar todas las pruebas necesarias para todos los posibles casos que se puedan presentar en el archivo de entrada, fue una tarea tediosa, pero que, de nuevo, conseguimos solucionar sin problemas.

En definitiva, una práctica de la que sacamos una gran cantidad de aprendizajes para poder aplicar tanto en conceptos teóricos de la asignatura como en futuras prácticas.