Introducción

En esta práctica intentaremos crear un lenguaje de programación que maneje sentencias de lógica de predicados y las resuelva. Construir un intérprete de expresiones lógicas es un proyecto de programación complejo y por ello, vamos seguir los siguientes pasos:

- Construir un programa (**analizador léxico**), que divida un texto en las unidades mínimas con significado (**tokens**);
- Construir el árbol de derivación, usando los tokens y la gramática (**analizador sintáctico**).

Empezaremos con el analizador léxico, poniendo el esqueleto y el resultado de la evaluación:

```
import re
import collections
from functools import wraps, partial

def debug_method(func= None, prefix = ''):
    if func is None:
        return partial(debug, prefix = prefix)
    else:
        msg = prefix + func.__name__
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(msg)
            return func(*args,**kwargs)
        return wrapper

def debug_class(cls):
    for key, val in vars(cls).items():
        if callable(val):
            setattr(cls, key, debug_method(val))
    return cls

print("Evaluación  del codigo")
separacion = 60
# Tipos de tokens
CONST     = r'(?P<CONST>[a-z][A-Z]*)'
NUM     = r'(?P<NUM>\d+)'
PLUS    = r'(?P<PLUS>\+)'
MINUS   = r'(?P<MINUS>-)'
OR    = r'(?P<OR>v)'
AND   = r'(?P<AND>∧)'
NOT   = r'(?P<NOT>¬)'
TIMES   = r'(?P<TIMES>\*)'
DIVIDE  = r'(?P<DIVIDE>/)'
LPAREN  = r'(?P<LPAREN>\()'
RPAREN  = r'(?P<RPAREN>\))'
WS      = r'(?P<WS>\s+)'
VERDADERO  = r'(?P<VERDADERO>TRUE)'
FALSO  = r'(?P<FALSO>FALSE)'

master_pattern = re.compile('|'.join((CONST,NUM, PLUS, MINUS, OR, AND, NOT,
                                      TIMES, DIVIDE, LPAREN, RPAREN, WS,
                                      VERDADERO, FALSO)))
Token = collections.namedtuple('Token', ['type', 'value'])


def lista_tokens(pattern, text):
    scanner = pattern.scanner(text)
    for m in iter(scanner.match, None):
        token = Token(m.lastgroup, m.group())

        if token.type != 'WS':
            yield token
print(list(lista_tokens(master_pattern,'x v y' )))
```

```
Evaluación  del codigo
[Token(type='CONST', value='x'), Token(type='OR', value='v'), Token(type='CONST', value='y')]
```

Aquí podemos ver los tres tokens que se generan. Son los que esperan, si la entrada es

`'x v y'`.

El analizador sintáctico es más complejo, ya que necesitamos definir una gramática.

La siguiente gramática tiene como variables *sentencia*, *conjuncion*, *clausula* y los símbolos terminales son:

```
(, ), ∧, ∨, CONST.
```

Las reglas de la gramática son:

```
sentencia : conjuncion | conjuncion ∨ sentencia
conjuncion : clausula | clausula ∧ conjuncion
clausula : CONST | (sentencia)
```

Como recordatorio, esta gramática se puede utilizar para ordenar las palabras, es decir, generarlas una por una mediante un programa.

```python
class GeneracionSentencias:
    def __init__(self, pos):
        self.pos = pos

    def sentencia(self):
        '''
        Sentencia retorna el string con la fórmula
        '''
        if self.pos % 2 == 0:
            self.pos = self.pos >> 1
            return self.conjuncion()
        else:
            self.pos = self.pos >> 1
            resultado = self.conjuncion()
            return  resultado+ "∨" + self.sentencia()


    def conjuncion(self):
        '''
        Sentencia retorna el string con la fórmula
        '''
        if self.pos%2 == 0:
            self.pos = self.pos >> 1
            return self.clausula()
        else:
            self.pos = self.pos >> 1
            resultado = self.clausula()
            return  resultado + "∧" + self.conjuncion()


    def clausula(self):
        '''
        Sentencia retorna el string con la fórmula
        '''
        if self.pos%2 == 0:
            self.pos = self.pos >> 1
            resultado, self.pos = chr(97 + (self.pos%25)), self.pos//25
            return resultado
        else:
            self.pos = self.pos >> 1
            return '(' + self.sentencia() + ')'
```

La implementación python que se puede generar, a partir de las reglas es la siguiente:

```python
class SentenciaBooleana:
    '''
    Pequeña implementación de un parser de sentenciaesiones booleanas.
    Implementation of a recursive descent parser.
    Aquí la asignacion es un diccionario con variables.
    '''
    def parse(self, text, asig):
          self.tokens = lista_tokens(master_pattern, text)
          self.current_token = None
          self.next_token = None
          self._avanza()
          self.asig = asig
          return self.sentencia()
```

```
    def _avanza(self):
            self.current_token, self.next_token = self.next_token, next(self.tokens, None)
    def _acepta(self, token_type):
            # if there is next token and token type matches
            if self.next_token and self.next_token.type == token_type:
              self._avanza()
              return True
            else:
              return False

    def _espera(self, token_type):
            if not self._acepta(token_type):
              raise SyntaxError('Expected ' + token_type)

    def sentencia(self):
            '''
            sentencia : conjuncion | conjuncion v sentencia
            '''
            sentencia_value = self.conjuncion()
            if self._acepta('OR'):
              sentencia_value = sentencia_value | self.sentencia()
            return sentencia_value

    def conjuncion(self):
            '''
            conjuncion : clausula | clausula ∧ conjuncion
            '''

            conj_value = self.clausula()
            if self._acepta('AND'):
              conj_value = conj_value & self.conjuncion()
            return conj_value

    def clausula(self):
            '''
            clausula : CONST | (sentencia)

            '''
            # Si aparece un parentesis
            if self._acepta('LPAREN'):
              sentencia_value = self.sentencia()
              self._espera('RPAREN')
              return sentencia_value
            elif self._acepta('CONST'):
              return self.asig[self.current_token.value]
e= SentenciaBooleana()
print(e.parse('x v (y ∧ y ∧ x)',{'x':False, 'y':False}))
```

```
False
```

Como vemos, esta implementación es **recursiva**, ya que tanto la construcción como el recorrido del árbol de derivación esta definida de forma recursiva.

Preguntas

1) *Crear una clase hija de **SentenciaBooleana**, llamada **SentenciaGeneral**, de forma que se admita el operador ¬ en la sentencia booleana, completando el siguiente código.*

```
class SentenciaGeneral(SentenciaBooleana):
    '''
    Pequeña implementación de un parser de sentencias booleanas.
    Aquí la asignacion es un diccionario con variables.
    '''
    def clausula_negativa(self):
        '''
        clausula :  ¬ CONST | ¬ (sentencia)

        '''
e = SentenciaGeneral()
print(e.parse('x v ¬(y ∧ ¬y ∧ x)',{'x':True, 'y':False}))
```

2) *Extender el código para generar palabras de la nueva gramática y guardar el número de la primera sentencia en la que aparezca la negación.*

3) *Crear un método que, dada una sentencia booleana, devuelva si es satisfactible y, en ese caso, un diccionario con una asignación que de verdadero. Explicar el porque se necesita hallar el cociente entre veinticinco.*

4) *Generar otra clase hija **SentenciaDot**, que genere un dibujo del árbol de derivación, utilizando el lenguaje dot. Aquí se pone un ejemplo*

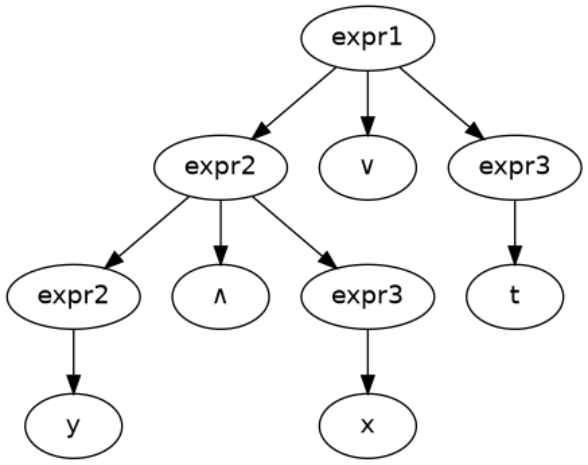

Figura 1: Ejemplo de figura y x t