Autómatas Y Lenguajes Formales

Curso 2023-2024

Autor : Mohammed Amrou Labied Nasser

DNI : 49857930W

Grupo: 2.2

Convocatoria: Enero

Profesor: Juan Antonio Sánchez Laguna

TABLA DE CONTENIDOS

1. …..... Manual del usuario.

2. …..... Aspectos principales.

2.1. …..... Menú principal.

2.2. …..... Función silabear.

2.3. …..... Función entonar.

4. …..... Conclusiones.

1.- Manual de usuario

La aplicación consiste en analizar una palabra de distintas formas utilizando expresiones regulares, para ello, existen tres opciones manejadas por un menú siendo elegidas tras introducir un numero (del 1 al 3). La primera opción es silabear (1) que trata en dividir las palabras en silabas siguiendo las reglas sintácticas de la lengua castellana, la siguiente opción es entonar (2) que muestra en qué lugar se sitúa la silaba tónica sustituyéndose la vocal por su correspondiente mayúscula. Y por último es exit (3) cuyo objetivo es salir de la aplicación.

2.- Aspectos Principales

En este apartado se analizarán las características más interesantes de la aplicación, así como el proceso de resolución mediante expresiones regulares de cada opción del menú. El desarrollo de la aplicación se ha realizado en un único modulo.

2.1 Menú principal

El menú principal consta de tres opciones como se mencionó previamente, cada una de las cuales realiza una tarea única y específica. A continuación, se muestra el código y una breve explicación pues se analizará cada apartado posteriormente de manera más profunda.

Primeramente, comprobamos si existe un fichero llamado `"dic.csv"` que contendrá palabras analizadas anteriormente, en caso de existir se lee el fichero y se le asocia sus claves y los valores al diccionario. Si el fichero no existe simplemente pasamos a la siguiente línea de código ya que este se creará posteriormente.

Seguidamente, se solicita una opción a elegir (un numero entero del 1 al 3) que en caso de introducir una opción fuera del rango se vuelve a solicitar la opción nuevamente.

Las opciones a elegir son tres, la primera es silabear donde se comprueba si existe la palabra en el diccionario, en caso de estar la palabra se devuelve su respectivo valor del diccionario, en caso de no existir se realiza el análisis de la palabra y se guarda en el diccionario. La segunda opción es entonar similar a silabear, solo se diferencia en que se devuelve la palabra con la vocal tónica en mayúscula. Y por último exit, cuya función es guardar las claves y valores de las nuevas palabras introducidas en el fichero `"dic.csv"`, para ello se utiliza .keys() obteniendo todas las claves y valores del diccionario que se guardan en el anterior fichero citado, hay que destacar la utilización de str() para cada valor, dado que el formato de la clave debe ser cadena y por poder adjuntar “;” a los valores de las claves, además otra funcionalidad es poder salir de la aplicación.

```
dicTod = dict()
def menu():
    try:
        f = open("dic.csv", "r", encoding="utf8")
        linea = f.readline()
        while linea:
            clave, valor1, valor2 = linea.strip().split(';')
            dicTod[clave] = [valor1, valor2]
            linea = f.readline()
        f.close()
    except FileNotFoundError:
        pass
    opcion = input("introduce opcion:\n 1 (silabear)\n 2
(entonar)\n 3 (exit)\n")
    try:
        opcion = int(opcion)
    except:
        print(f"{opcion} no es una opcion")
    if opcion == 1:
        palabra = input("Introduce una palabra para silabear:\n")
        palabra = palabra.lower()
        if palabra in dicTod:
            print("El resultado ya se encuentra en el diccionario
y es: ", dicTod[palabra][0])
        else:
            print("Su silabeo es: ", silabear(palabra))
            dicTod[palabra] = [silabear(palabra),
entonar(palabra)]
    elif opcion == 2:
        palabra = input("Introduce una palabra para clasificar
según su entonación:\n")
        palabra = palabra.lower()
        if palabra in dicTod:
            print("El resultado ya se encuentra en el diccionario
y es : ", dicTod[palabra][1])
        else:
            print("El resultado es: ", entonar(palabra),
clasifica(palabra))
            dicTod[palabra] = [silabear(palabra),
entonar(palabra)]
    elif opcion == 3:
        fsal = open("dic.csv", 'w', encoding="utf8")
        for clave in dicTod.keys():
            valores = dicTod.get(clave)
            valores_cadena = [str(valor) for valor in valores]
```

```
            print(clave, ";".join(valores_cadena), file=fsal,
sep=";")
        fsal.close()
        print("Adios!")
        exit(0)
```

2.2 Función de silabear

Esta función es de especial importancia ya que es la encargada de dividir las palabras y sirve como soporte en la función entonación. Para su realización se definieron las siguientes expresiones regulares, donde todas ellas se han unido en una única regla con la disyunción.

```
R1 =
r'(?P<R1>[aeiouáéíóú]([bcdfgjklmnñpqrstvwxyz]|ll|rr|ch)[aeiouüáéió
úy])'
R2a = r'(?P<R2a>[aeiouáéíóú][pcbgf][rl][aeiouáéíóúy])'
R2b = r'(?P<R2b>[aeiouáéíóú][dt][r][aeiouáéíóúy])'
R2c =
r'(?P<R2c>[aeiouáéíóú][bcdefghjklmnñpqrstvwxyz][bcdefghjklmnpqrstv
wxyz][aeiouáéíóúy])'
R3a =
r'(?P<R3a>[aeiouáéíóú][bcdefghjklmnñpqrstvwxyz](([pcbgf][rtl])|([d
t][r]))[aeiouáéíóúy])'
R3b =
r'(?P<R3b>[aeiouáéíóú][bdnmlr][s][bcdefghjklmnñpqrstvwxyz][aeiouáé
íóúy])'
R3c =
r'(?P<R3c>[aeiouáéíóú][s][t][bcdefghjklmnñpqrstvwxyz][aeiouáéíóúy]
)'
R4a =
r'(?P<R4a>[aeiouáéíóú](([bdnmlr][s]|[s][t]))[pcbgf][rl][aeiouáéíóú
y])'
''' # sin ataque
R5a1 = r'(?P<R5a1>[aeoáéó]h?[iu])')
R5a2 = r'(?P<R5a2>[iu]h?[aeoáéó])'
R5a3 = r'(?P<R5a3>(([ií]h?[uúü])|([úuü]h?[ií])))'
'''
R5b1 = r'(?P<R5b1>(([aeo]h?[úí])|([úí]h?[aeo])))'
R5b2 = r'(?P<R5b2>[aá][aá]|[eé][ée]|[ií][íi]|[oó][oó]|[uú][úu])'
R5b3 = r'(?P<R5b3>(([aá]|[eé]|[oó])([oó]|[aá]|[eé])))'
R5c = r'(?P<R5c>([aeiouáéíóú]h[aeiouáéíóúy]))'
R6a = r'(?P<R6a>[iu][aeoáéó][iuy])'
```

Para implementar el algoritmo de división se ha utilizado dos variables que almacenan el corte anterior y el corte actual de la palabra, esta primera se actualiza en cada iteración siempre que la longitud de la palabra restante sea mayor al corte actual y que exista alguna expresión regular posible de aplicar. Para determinar el corte actual en cada expresión regular se le suma un determinado número entero dependiendo de las características de la expresión y se añade a la lista, la palabra desde el corte anterior hasta el corte actual y se actualiza el valor del corte anterior siendo ahora el del corte actual.

Un ejemplo sería la palabra amigo donde se identifica la R1 debido al patrón “ami” y se le suma uno siendo la palabra de la siguiente forma “a” “migo” y se procesa ahora solo la subpalabra “migo”, nuevamente se localiza el patrón “igo” de la R1 y se procede a sumarle uno siendo ahora la lista así, [“a”,”mi”]. Como la última subpalabra no coincide con ninguna expresión regular se termina la iteración y se mete el resto en la lista siendo el resultado así, [“a”,”mi”, “go”]. Hay casos donde se identifica expresiones regulares que no necesitan sumarles nada ya que son reglas para silabas que no tienen ataque. También es llamativo ver que en la última sección del código se comprueba si el último elemento de la lista es una cadena vacía, ya que en ocasiones para las palabras de la regla 6 se inserta una cadena vacía por lo que es importante que sea eliminada. Finalmente, se devuelve la nueva lista, a continuación, se muestra el código entero sin las expresiones regulares.

```
def silabear(cadena):
     # EXPRESIONES REGULARES

    R = "(?i)" + R1 + "|" + R2a + "|" + R2b + "|" + R2c + "|" +
R3a + "|" + R3b + "|" + R3c + "|" + R4a + "|" + R5c + "|" + R6a  +
"|" + R5b1 + "|" + R5b2 + "|" + R5b3
    er = re.compile(R)
    corteAnterior = 0
    silabas = []
    while corteAnterior < len(cadena):
        m = er.search(cadena, corteAnterior)
        if not m:
            break
        if m.group("R1"):
            corteActual = m.start() + 1
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R2a"):
            corteActual = m.start() + 1
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
```

```
        elif m.group("R2b"):
            corteActual = m.start() + 1
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R2c"):
            corteActual = m.start() + 2
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R3a"):
            corteActual = m.start() + 2
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R3b"):
            corteActual = m.start() + 3
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R3c"):
            corteActual = m.start() + 3
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R4a"):
           corteActual = m.start() + 3
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        # sin ataque R5a1, R5a2, R5a3
       elif m.group("R5b1"):
            corteActual = m.start() + 1
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R5b2"):
            corteActual = m.start() + 1
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R5b3"):
            corteActual = m.start() + 1
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R5c"):
            corteActual = m.start() + 1
            if corteActual > corteAnterior:
```

```
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
        elif m.group("R6a"):
            corteActual = m.start() + 4
            if corteActual > corteAnterior:
                silabas.append(cadena[corteAnterior:corteActual])
                corteAnterior = corteActual
    silabas.append(cadena[corteAnterior:])
    if silabas[-1] == "":
        silabas.pop()
    return silabas
```

2.3 Función Entonar

Esta función es la encargada de marcar la vocal tónica sustituyéndola por su respectiva mayúscula. De igual manera existe otra función llamada clasifica cuyo objetivo es ordenar las palabras según su tipo, es decir si son agudas, llanas, esdrújulas o sobresdrújulas ya que entonar no realiza esa labor, su implementación sigue el siguiente patrón:

```
def clasifica(palabra):
    Rmay = r'(?P<Rmay>[AEIOU])'
    may = re.compile(Rmay)
    en = entonar(palabra)
    if en:
        ultima = en[-1]
        m = may.search(ultima)
        if m:
            return " y es aguda"
        if len(en) > 1:
            penultima = en[-2]
            m1 = may.search(penultima)
            if m1:
                return " y es llana"
        if len(en) > 2:
            antepenultima = en[-3]
            m2 = may.search(antepenultima)
            if m2:
                return " y es esdrújula"
        if len(en) > 3:
                return " y es sobreesdrújula"
```

En primer lugar, debemos destacar que la función entonar devuelve una lista de silabas cuya vocal tónica está en mayúscula, en consecuencia, basta con definir una expresión regular de las vocales en mayúsculas y comprobar que el último elemento de la lista tiene una vocal en mayúscula en el caso de que fuera aguda, realizar lo

mismo con el penúltimo elemento de la lista, además de verificar que su longitud es mayor que uno en el caso de que fuera llana y efectuar el mismo patrón sucesivamente para las distintas alternativas.

Por otro lado, la implementación de la función entonar sigue la misma lógica que silabear puesto que se unen las reglas gramaticales para identificar las silabas tónicas en una única regla. El procedimiento para identificar una subregla es similar al usado en silabear, comprobando primeramente si lleva tilde y posteriormente aplicando la función tilde, después se verifica si es aguda y se aplica su respectiva función y finalmente si es llana. El código es el siguiente;

```
def entonar(palabra):
    Rtildes = r'(?P<Rtildes>[áéíóú])'
    Raguda = r'(?P<aguda>[^nsaeiou]$)'
    Rllana = r'(?P<llana>[nsaeiou]$)'

    det = "(?i)" + Rtildes + "|" + Raguda + "|" + Rllana
    er = re.compile(det)

    m = er.search(palabra)

    def tilde(palabra):
       # se muestra despues

    def aguda(lista_de_silabas):
      # se muestra despues

    def llana(lista_de_silabas)
      # se muestra despues


    if m.group("Rtildes"):
        return silabear(tilde(palabra))
    elif m.group("aguda"):
        return aguda(silabear(palabra))
    else:
        return llana(silabear(palabra))
```

Es evidente que la función `tilde(palabra)` tendrá la misma lógica que silabear y entonar, lo único que difiere es que se debe sustituir la vocal con tilde por su respectiva mayúscula para ello se utiliza `sub()`que es un mecanismo de sustitución aplicada a las expresiones regulares. Algo que destacar es que no se silabea la palabra antes de aplicar la función ya que resulta menos complejo.

```
def tilde(palabra):
    Ra = r'(?P<Ra>á)'
    Re = r'(?P<Re>é)'
    Ri = r'(?P<Ri>í)'
    Ro = r'(?P<Ro>ó)'
    Ru = r'(?P<Ru>ú)'
    Rtildes = "(?i)" + Ra + "|" + Re + "|" + Ri + "|" + Ro + "|" +
Ru
 re_tildes = re.compile(Rtildes)
    m = re_tildes.search(palabra)
if m.group("Ra"):
        sust = re_tildes.sub('A', palabra)
        return sust
    elif m.group("Re"):
        sust = re_tildes.sub('E', palabra)
        return sust
    elif m.group("Ri"):
        sust = re_tildes.sub('I', palabra)
        return sust
    elif m.group("Ro"):
        sust = re_tildes.sub('O', palabra)
        return sust
    elif m.group("Ru"):
        sust = re_tildes.sub('U', palabra)
        return sust
```

En el caso de la función aguda sí que es importante silabear la palabra para poder operar solamente con la última silaba, el mecanismo de sustitución sigue realizándose con sub() y es importante poner el count=1 ya que en el caso de que fuera diptongos la sustitución se realiza para ambas vocales. Al final se devuelve la lista original con el último elemento modificado.

```
def aguda(lista_de_silabas):
    Ra = r'(?P<Ra>a)'
    Re = r'(?P<Re>e)'
    Ri = r'(?P<Ri>i)'
    Ro = r'(?P<Ro>o)'
    Ru = r'(?P<Ru>u)'
    Rvocal = "(?i)" + Ra + "|" + Re + "|" + Ri + "|" + Ro + "|" +
Ru
    Rvocales = re.compile(Rvocal)

    ultima_silaba = lista_de_silabas[-1]
    m = Rvocales.search(ultima_silaba)
    if m:
        vocal_con_tilde = m.group()
```

```
        vocal_mayuscula = vocal_con_tilde.upper()
        sust = Rvocales.sub(vocal_mayuscula, ultima_silaba,
count=1)
        return lista_de_silabas[:-1] + [sust]

    return lista_de_silabas
```

La función llana seguirá el mismo patrón que la aguda, solamente que se realizará la modificación sobre el penúltimo elemento de la lista y se devolverá la lista original con la modificación sobre el penúltimo elemento.

```
Ra = r'(?P<Ra>a)'
Re = r'(?P<Re>e)'
Ri = r'(?P<Ri>i)'
Ro = r'(?P<Ro>o)'
Ru = r'(?P<Ru>u)'
Rvocal = "(?i)" + Ra + "|" + Re + "|" + Ri + "|" + Ro + "|" + Ru
Rvocales = re.compile(Rvocal)

penultima_silaba = lista_de_silabas[-2]
m = Rvocales.search(penultima_silaba)

if m:
    vocal_con_tilde = m.group()
    vocal_mayuscula = vocal_con_tilde.upper()

    sust = Rvocales.sub(vocal_mayuscula, penultima_silaba,
count=1) # count=1 porque sino sustituye todo
    return lista_de_silabas[:-2] + [sust] + [lista_de_silabas[-1]]

return lista_de_silabas
```

2.3 Conclusión y valoración personal

Durante la realización de este proyecto, he aprendido que numerosos problemas se pueden resolver aplicando expresiones regulares disminuyendo el trabajo de desarrollo y la dificultad en comparación con una implementación tradicional (usando bucles, sentencias, etc.) A lo largo de las sesiones he ido ganando experiencia en la creación y aplicación de expresiones regulares para abordar problemas específicos que se han ido proponiendo en la aplicación.

Respecto a mi opinión, el uso de expresiones regulares es algo fundamental para cualquier informático puesto que es una manera de agilizar y acortar el tiempo empleado para el desarrollo de código. Además, numerosos intérpretes de comandos (bash) hacen uso de elementos propios de las expresiones regulares debido a que proporciona una forma poderosa y flexible de trabajar con textos de manera eficiente y efectiva, realizando tareas como la búsqueda y filtración de ficheros o directorios, comodines, manipulación de códigos directamente desde la línea de comandos.

Por último, he de destacar que no se ha realizado la entonación de triptongos y diptongos de forma específica ya que en el castellano en numerosas ocasiones estas vocales pueden ser separadas, además de que suelen ir acentuadas, por otro lado, la clasificación de las palabras solo se realiza si esta no está contenida en el diccionario.