

Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación.

IC5701 – Compiladores e Intérpretes.

Proyecto 2 Explorador – Ciruelas Extendido

Profesor: Aurelio Sanabria.

Estudiante:

Tamara Nicole Rodríguez Luna.

19 de abril del 2025.

Demostración y discusión

En Ciruelas el explorador no almacenaba la línea y columna de los componentes, por lo tanto, los agregué para poder mostrar errores más precisos, de paso adaptando el comentario para que esté acorde con el código

```
339dd9 - Agregar línea y col... (1 file) x
explorador.py explorador
29 hidden lines
30 """
31 Clase que almacena la información de un componente léxico
32
33 Notese que no almacena información auxiliar para mostrar errores lo
34 cuál es terrible
35 """
36
37 tipo : TipoComponente
38 texto : str
39
40 def __init__(self, tipo_nuevo: TipoComponente, texto_nuevo: str):
41     self.tipo = tipo_nuevo
42     self.texto = texto_nuevo
43
44 def __str__(self):
45 """
46 Da una representación en texto de la instancia actual usando un
47 string de formato de python (ver 'python string formatting' en
48 google)
29 hidden lines
30 """
31 Clase que almacena la información de un componente léxico
32
33+ Almacena información auxiliar para mostrar errores (línea y columna)
34 """
35
36 tipo : TipoComponente
37 texto : str
38+ linea : int
39+ columna : int
40
41+ def __init__(self, tipo_nuevo: TipoComponente, texto_nuevo: str, linea:
42     self.tipo = tipo_nuevo
43     self.texto = texto_nuevo
44+     self.linea = linea
45+     self.columna = columna
46
47 def __str__(self):
48 """
49 Da una representación en texto de la instancia actual usando un
50 string de formato de python (ver 'python string formatting' en
51 google)
```

Seguidamente añadí los nuevos componentes léxicos de la gramática que definí para ciruelas

```
dc5f386 - Agregar nuevos compo... (1 file) x
explorador.py explorador
23 hidden lines
24 PUNTUACION = auto()
25 BLANcos = auto()
26 NINGUNO = auto()
27
28
29 class ComponenteLéxico:
38 hidden lines
68 descriptores_componentes = [
69     (TipoComponente.COMENTARIO, r'^#Bomba:#'),
70     (TipoComponente.PALABRA_CLAVE, r'^((mae|sarpe|jere|jeña|satis))'),
71
72     (TipoComponente.CONDICIONAL, r'^((diay siii|sino ni modo)'), 
73     (TipoComponente.REPETICION, r'^((ueee)'), 
74     (TipoComponente.ASIGNACION, r'^((metale)'), 
75
76     (TipoComponente.PALABRA_CLAVE, r'^((mae|sorpe|jeñe|jeña|satis)'), 
77     (TipoComponente.SWITCHCASE, r'^((como está la vara)'), 
78     (TipoComponente.CASE, r'^((movida)'), 
79     (TipoComponente.TRY, r'^((jueguesela)'), 
80     (TipoComponente.CATCH, r'^((tortón)'), 
81     (TipoComponente.INVOCACION, r'^((lameeee)'), 
82     (TipoComponente.FOR, r'^((deja vuelta)'), 
83     (TipoComponente.CONDICIONAL, r'^((diay siii|sino ni modo)'), 
84     (TipoComponente.REPETICION, r'^((ueee)'), 
85     (TipoComponente.ASIGNACION, r'^((metale)'), 
86
87
88 95 hidden lines
```

Luego creé una clase ErrorExplorador para poder almacenar los posibles errores que puede dar el explorador, así puedo irlos almacenando en una lista e imprimir todos los errores cuando el explorador termine

```

63
64+class ErrorExplorador:
65+    """
66+        Clase que almacena errores léxicos encontrados por el explorador
67+
68+    def __init__(self, mensaje, texto, linea, columna):
69+        self.mensaje = mensaje
70+        self.texto = texto
71+        self.linea = linea
72+        self.columna = columna
73+
74+    def __str__(self):
75+        return f"ERROR: {self.mensaje} en linea {self.linea}, columna {self.columna} - Texto: '{self.texto}'"
76+
77 class Explorador:
78     """
79     Clase que lleva el proceso principal de exploración y deja listos los
80     * 30 hidden lines
81     def __init__(self, contenido_archivo):
82         self.texto = contenido_archivo
83         self.componentes = []
84         self.errores = []
85
86     def explorar(self):
87         """
88
89         self.componentes = self.componentes + resultado
90         indice_linea += 1
91
92         # Si se encontró 1 o más errores se imprimen y se detiene el programa
93         if len(self.errores) > 0:
94             self.imprimir_errores()
95             sys.exit()
96
97         def imprimir_componentes(self):
98             """
99                 Imprime en pantalla en formato amigable al usuario los componentes
100
101         4 hidden lines
102         print(componente) # Esto funciona por que el print llama al
103                         | # método __str__ de la instancia
104
105         def imprimir_errores(self):
106             """
107                 Imprime todos los errores encontrados por el explorador en una estructura legible para el programador
108             """
109             print("\n==== ERRORES ENCONTRADOS ====")
110             for error in self.errores:
111                 print(error)
112             print(f"Total de errores: {len(self.errores)}")
113
114         def registrar_error(self, mensaje, texto, linea, columna):
115             """
116                 Registra un error para ser mostrado al final de la exploración
117             """
118             error = ErrorExplorador(mensaje, texto, linea, columna)
119             self.errores.append(error)
120
121         def procesar_linea(self, linea, indice_linea):
122             """
123

```

El único error que pude identificar es cuando el carácter no corresponde a ninguno de los ya establecidos

Por alguna razón, Ciruelas original me estaba dando error al tratar de procesar los caracteres con tildes o diéresis o la ñ, para solucionar eso hice que re funcione en unicode, agregué la codificación utf-8 y todos los caracteres especiales del español a las expresiones regulares, es algo que ya había contemplado en mi documento de gramática. Por otro lado, en este mismo commit, vi que el orden de las expresiones regulares importa, tuve que mover la expresión regular de los flotantes antes de los enteros, sino todos los números los detectaba como enteros.

028c2ff - Se arregla el manejo... (3 files) ×

• ciruelas.py

6 hidden lines

```
7 from generador.generador import Generador
8
9 import argparse
```

77 hidden lines

• explorador.py explorador

99 hidden lines

```
100 (TipoComponente.OPERADOR, r"^(echelle|quitele|chunchereque|desmadeje|divorcio|casorio)",),
101 (TipoComponente.COMPARADOR, r"^(calazo|poquito|mismavara|otravara|menos o igualitico|más o igual"),
102 (TipoComponente.TEXTO, r"^(.-|?|^*)"),
103 (TipoComponente.IDENTIFICADOR, r"^(a|b|c|(a-zA-Z0-9)*")"),
104 (TipoComponente.ENTERO, r"^-?(\d+|\d+.\d+)"),
105 (TipoComponente.FLOTANTE, r"^-?(\d+\.\d+|\d+\d+)"),
106 (TipoComponente.VALOR_VERDAD, r"^(True|False)"),
107 (TipoComponente.PUNTIACION, r"^(.|\(|\)|\)|\.)"),
108 (TipoComponente.BLANCOS, r"^(s+| )")
```

110 def __init__(self, contenido_archivo):
111 self.texto = contenido_archivo
112 self.componentes = []

113
114 self.erroros = []

115
116 def explorar(self):
117 """ Itera sobre cada una de las líneas y las va procesando de forma que

113 hidden lines

• archivos.py util

5 hidden lines

```
6 en caso de que el archivo
7 """
8
9     with open(ruta_archivo) as archivo:
10         for linea in archivo:
11             yield linea.strip("\n")
```

14

• 6 hidden lines

```
7 from generador.generador import Generador
8
9 import argparse
10 import sys
11 import io
12
13# Se configura la salida estandar para manejar UTF-8
14sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8')
15sys.stderr = io.TextIOWrapper(sys.stderr.buffer, encoding='utf-8')
16
17 parser = argparse.ArgumentParser(description='Interprete para Ciruelas (el lenguaje)')
18
```

77 hidden lines

• 99 hidden lines

```
100 (TipoComponente.OPERADOR, r"^(echelle|quitele|chunchereque|desmadeje|divorcio|casorio)",),
101 (TipoComponente.COMPARADOR, r"^(calazo|poquito|mismavara|otravara|menos o igualitico|más o igual"),
102 (TipoComponente.TEXTO, r"^(.-|?|^*)"),
103 (TipoComponente.IDENTIFICADOR, r"^(a|b|c|(a-zA-Z0-9)*")"),
104 (TipoComponente.FLOTANTE, r"^-?(\d+\.\d+|\d+\d+)"), # Detectar los flotantes antes para que no se confundan con enteros
105 (TipoComponente.ENTERO, r"^-?(\d+|\d+.\d+)"),
106 (TipoComponente.VALOR_VERDAD, r"^(True|False)"),
107 (TipoComponente.PUNTIACION, r"^(.|\(|\)|\.)"),
108 (TipoComponente.BLANCOS, r"^(s+| )")
```

109
110 def __init__(self, contenido_archivo):
111 self.texto = contenido_archivo
112 self.componentes = []
113
114 # Lista de errores encontrados
115 self.erroros = []
116
117 # Se asegura que regex funcione en modo unicode para soportar caracteres especiales
118 re.UNICODE
119
120 def explorar(self):
121 """
122 Itera sobre cada una de las líneas y las va procesando de forma que
123
124 113 hidden lines

• 5 hidden lines

```
6 en caso de que el archivo
7 """
8
9     with open(ruta_archivo, "r", encoding="utf-8") as archivo:
10         for linea in archivo:
11             yield linea.strip("\n")
```

14

Ciruelas es un lenguaje que me parece muy similar a Python en su ausencia de ; o algún otro símbolo para terminar una línea, aún así Ciruelas sigue utilizando las llaves para delimitar una función, por ejemplo, lo que en Python veríamos así:

```
def imprimir(texto):
```

```
print(texto)
```

En Ciruelas se vería así:

```
mae imprimir (texto){  
    sueltele(texto)  
}
```

Esto entiendo que es por la simplicidad de buscar un símbolo específico para saber que ya terminó una función, hacer eso es mucho más seguro que el manejo de indentación que hace Python. Pero al centrarnos en las funciones nuevas que estoy proponiendo en Python tenemos el try-except que se ve así:

```
try:  
    resultado = a / b  
except:  
    print("No puedes dividir por cero")
```

Mientras que en Ciruelas mi propuesta es que se vea así, es menos limpio con respecto a cantidad de caracteres, pero más fácil de procesar:

```
juéguesela{  
    resultado metale a desmadeje b  
}tortón{  
    sueltele(~No puedes dividir por cero~)  
}
```

Además en Python hay un for que lo podemos ver en el siguiente ejemplo:

```
for i in range(len(texto)):  
    print(texto[i])
```

El de Ciruelas que propongo pienso que es bastante diferente, pero queda más claro todas las variables involucradas y el "step" del for, que en este caso sube de 1 en 1:

```
dele vuelta (i metale 0 / i poquitico largo_texto / i metale i  
hechele 1){  
    sueltele(viene_bolita(texto / i))  
}
```

Por último, el switch-case es extraño de comparar puesto que nada similar existía antes de Python 3.10, es curioso que se tardara tanto en implementar, sí técnicamente no hace falta, pero muchas veces se termina viendo más limpio que otras opciones como solo implementaciones alternativas de tratar de hacer lo mismo únicamente con if, diccionarios o funciones, luego se implementó el match que tiene la siguiente forma:

```
match x:  
    case 1:  
        print("El resultado es 1")  
    case 2:  
        print("El resultado es 2")  
    case _:  
        print("El resultado es diferente a 1 y 2")
```

En la extensión de Ciruelas yo propuse que tuviera una forma similar, pues no me agradan los breaks que se pueden encontrar en los switches de otros lenguajes, pero con más llaves para saber dónde empieza y dónde termina cada caso, como se muestra a continuación:

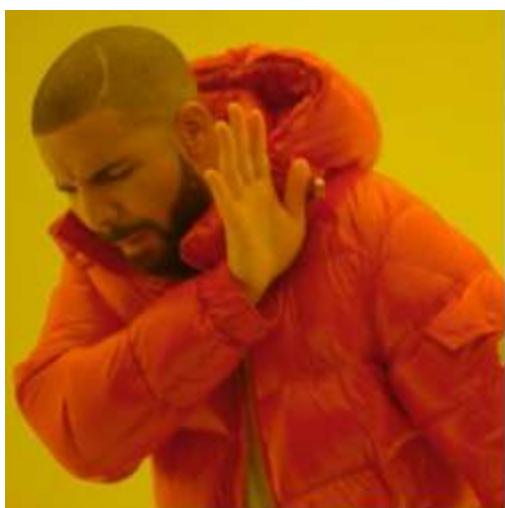
```
como está la vara(x){  
    movida 1{  
        sueltele(~El resultado es 1~)  
    }  
    movida 2{  
        sueltele(~El resultado es 2~)  
    }  
    sino ni modo{  
        sueltele(~El resultado es diferente a 1 y 2~)  
    }  
}
```

Lecciones aprendidas

Lo que voy aprendiendo de esto es primeramente no fiarse del código ya hecho, no hay que optimizar absolutamente todo si no hace falta para evitar un problema mayor, pero el caso de los caracteres especiales del español cuando vi que no se procesaron supe que tenía que

modificar lo que ya estaba hecho, igualmente cuando vi que el explorador asumía que todos los números eran enteros únicamente porque esa definición estaba antes que la de flotante, entiendo que eso pasa porque el intérprete de Python funciona de arriba hacia abajo y como vio un match no se molestó en verificar los demás. Por último, me sorprendió que el manejo de columnas fuera tan raro, pensé que era un estandar, pero cuando vi las inconsistencias entre la columna que me indicaba Visual Studio Code y el block de notas pude aprender que cada editor de texto considera al tabulador de formas diferentes, algunos lo ven como solo 1 caracter, otros como 2, 4 u 8 caracteres, esa era la razón, entonces tuve que decidir un número que fue 4, principalmente para seguir lo que indica VS Code.

Memes



**Python
revisando todas
las expresiones
regulares**



**Python siendo
un vago y
tomando la
primera que sirve**



YO

JAKE-CLARK.TUMBLR

