Actividad Integradora 3.2 Resaltador de sintaxis

Implementación de métodos computacionales

Grupo 604

Equipo:

Eugenio Garza	**A00836687**
Fernando Pérez García	**A01285236**
Luis Guillermo Blackaller	**A01178173**
Hervey Navarro Olazarán	**A01178086**

Fecha de entrega:

16 abr 2024

Reporte

Un parser es un analizador sintáctico el cual sirve como herramienta fundamental para el proceso de análisis de lenguajes de programación. Tiene como objetivo interpretar la estructura sintáctica de un texto dependiendo de las reglas que se establezcan en la gramática del lenguaje. El parser lo que hace es verificar si la secuencia de tokens generada por el escáner sigue las reglas de la gramática del lenguaje. El token es utilizado como una unidad de significado el cual el analizador léxico identifica y clasifica en el texto de entrada. Dentro de él existen los tokens, los cuales representan elementos como palabras clave, ids, operadores, símbolos y otros más. Cada Token tiene asociado un tipo que indica su categoría dentro del mismo lenguaje.

Lo que hicimos en esta actividad fue dividir el trabajo en diferentes secciones las cuales llevarán una forma de escalera, que va subiendo conforme más escalones vamos avanzando. Primero empezamos con el autómata finito determinista. Este autómata es un modelo matemático el cual es utilizado para la computación y la teoría de la computación para representar sistemas de cómputo con comportamientos específicos. Diseñamos este AFD con el fin de que pueda reconocer los elementos de léxico del lenguaje que se nos ha sido solicitado. Después de esto, continuamos con la creación de un analizador léxico que siguiera la técnica de la matriz de transiciones, nombrado obten_token.py. También creamos un analizador de sintaxis usando el método de descenso recursivo llamado partner.py. Ambos de estos documentos fueron entregados por el profe, solo hacían faltas algunas modificaciones para que los códigos pudieran cumplir con los requisitos.

La funcionalidad del programa es leer las expresiones que se establecen desde el teclado. Con eso en mente, el programa deberá realizar un análisis léxico y sintáctico sobre la misma entrada y al mismo tiempo generar un código HTML. Lo que incluirá este código, serán los elementos que fueron identificados mediante una variedad de colores diferentes, uno distinto para cada token. Esto le crea un resultado más ilustrativo y más estructurado a la solución, ya que mantendrá una diferencia en cada símbolo y así será más fácil identificarlos directamente.

En cuanto a la dificultad del trabajo, se puede pensar que hay muchos factores que influyen en todas sus dificultades. Por un lado, diseñar e implementar AFD para identificar los componentes léxicos de una lengua requiere un análisis cuidadoso de la gramática y del significado correctamente transmitido de cada símbolo. Este proceso puede resultar difícil, dependiendo del tamaño y la complejidad de la gramática involucrada. Por otro lado, la implementación de verificadores de texto y sintaxis agrega complejidad, especialmente cuando se considera la generación de código HTML y el análisis de texto. Si bien no es muy difícil, combine su producto con código HTML de manera consistente y efectiva. Además, la corrección de errores (como encontrar y clasificar errores léxicos o sintácticos) aumenta la complejidad de la tarea.

En resumen, este trabajo incluye una variedad de funciones complejas, desde el diseño del AFD hasta la implementación de editores de texto y gramática, así como la creación de código HTML y texto plano. La mala gestión y coordinación entre las diferentes partes de un proyecto pueden causar todas las dificultades en un proyecto que puede completarse con éxito a pesar de una buena gestión y planificación.

Conclusiones Individuales

Eugenio:

En esta actividad, creamos un resaltador de sintaxis en el cual utilizamos un código llamado obten_token.py y parser.py. En obten_token.py creamos un número determinado de tokens los cuales utilizamos para representar los diferentes factores dentro del código. Cada uno tenía su propio propósito y representaba un factor diferente dentro de una oración. El parser lo utilizamos para simplificar las estructuras y poder tener un trabajo con menor complejidad. Dentro del parser se crea un archivo HTML en el cual se despliega la oración compuesta quebrada en los diferentes tokens determinados. Para esto, utilizamos un archivo .css que utilizamos para poder colorear los tokens de colores distintos. Para mi lo mas dificil del proyecto fue la creación de la matriz dentro del archivo de de obten_token. Esta parte del proyecto la considero la más complicada ya que fue con lo que más batallamos mi equipo y yo. Esto se debe a que hubo demasiada confusión dentro del planteamiento por las diferentes opiniones que platicamos como equipo. Al final decidimos hacerlo de la manera que nos pareció más efectiva y los resultados reflejan esa efectividad. Lo más fácil pero no menos importante fue el archivo css en el que agregamos los diferentes colores que portaban los diferentes símbolos dependiendo su token. El orden de complejidad del código que implementamos es O(n). Para el código HTML, se tuvo que programar en el código del parser para que se pudiera crear uno nuevo cada vez que se corriera el código sin errores . Todo funcionó exitosamente y pudimos completar con todos los requerimientos del proyecto.

Fernando:

En conclusión, el algoritmo planteado fue un poco más eficiente que la solución original planteada por la plantilla del parser, esto debido a que la solución original mandaba a llamar el scanner cada que terminaba de determinar si el token era correcto o bien valido. En cambio nosotros vimos la oportunidad de hacerlo de una manera más eficiente al leer todo el input desde un inicio y en vez de regresar tokens con la función de obten_token() regresamos un arreglo con todos los tokens identificados en el input. Y cada que mandábamos a llamar match, si se cumplía la condición, simplemente incrementamos en uno en el índice del arreglo en el que nos

encontramos actualmente para poder seguir analizando la sintaxis con los tokens. Lo más complicado para mí fue poder echar a andar el parser, ya que el lenguaje planteado no se entendía muy fácilmente. Lo más fácil fue hacer la función de obten_token() ya que nos podíamos apoyar un poco más en la lógica de programación y no tanto en un autómata como en jflap, a pesar de que al final del día lo que hicimos fue un autómata. La complejidad del código es de O(2) ya que tanto obten_token() tiene una complejidad de O(n), y parser solamente recibe el arreglo de tokens y obten_token() tiene una complejidad de O(n). Para generar el código en HTML escribimos en un archivo html vacio los headers y todo lo necesario para empezar al inicio de parser(). Después cada que obten_token() identificaba un token, escribimos en el archivo html con su tag y su clase de css. Así sucesivamente hasta llegar al final o a un error. En el caso de error manejamos una función que recibe mensajes de error dependiendo de qué tipo de error sea, y lo despliega en el html y cerramos todos los tags que abrimos en un inicio. Si no hay error, la función de parser llega al final, en el cual cerramos todos los tags al igual que en el caso de error. Al final todo funcionó a la perfección, gracias a una combinación de un obten token que funciona a la perfección y una gramática que nos permite escribir adecuadamente o reconocer errores.

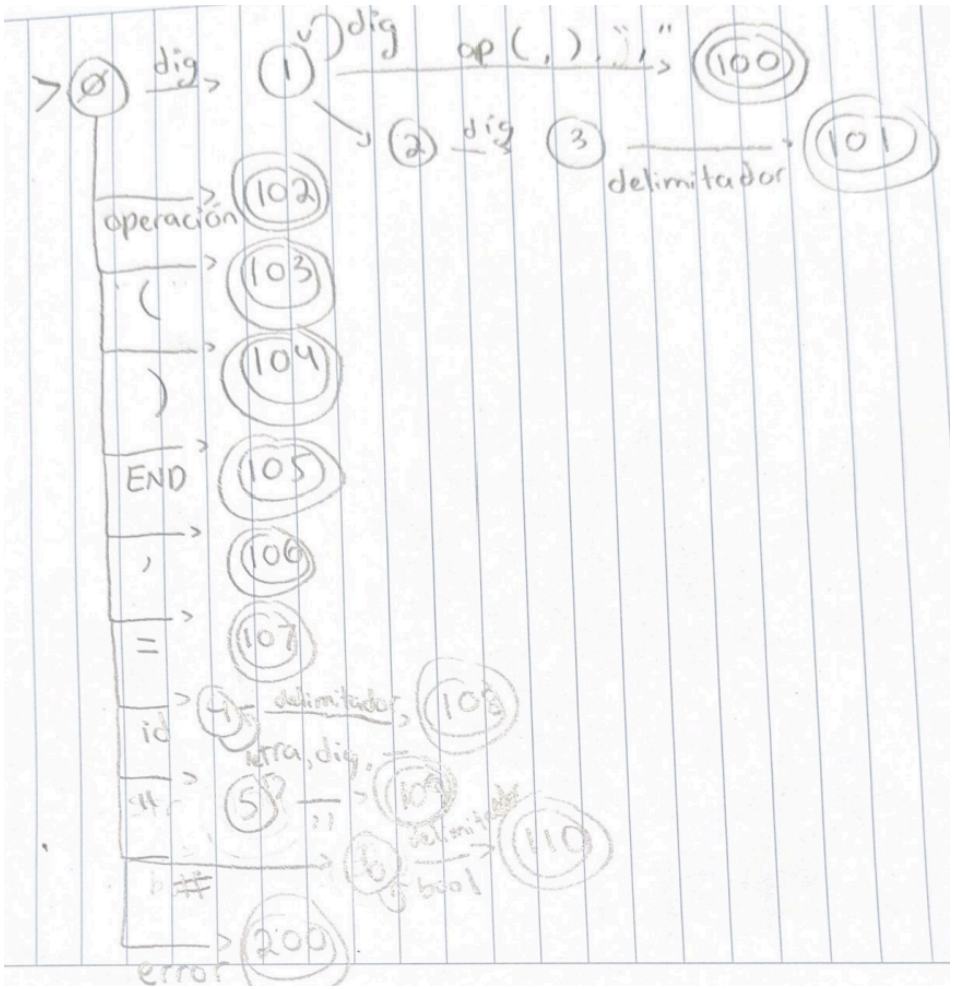
Hervey:

Durante este proyecto, me encontré con algunos problemas, pero también aprendí muchas cosas nuevas que me ayudaron a ser mejor en la programación. Tratamos de hacer un programa que pudiera resaltar diferentes partes de un código. Lo más difícil para mí fue entender cómo hacer que el programa reconociera las diferentes partes del código y las coloreara correctamente. Trabajando con mis amigos, pudimos entenderlo y hacerlo bien. También tuve problemas para hacer que el programa creara un archivo HTML al final. Al principio, no estaba seguro de cómo hacerlo, pero resultó ser más fácil de lo que pensaba. Este proyecto me enseñó la importancia de trabajar juntos y de no rendirse cuando las cosas se ponen difíciles. Aunque cometí errores y encontré problemas, aprendí de ellos y seguí intentándolo hasta que funcionó. En resumen, este proyecto me ayudó a mejorar mis habilidades en programación y me enseñó la importancia de trabajar en equipo y ser persistente.

Luis:

En este proyecto desarrollamos un resaltador de sintaxis con dos programas, divide el código en bloques y los colorea según el tipo identificado. Un programa genera tokens para representar las partes del código y otro para simplificar la estructura de estos. El output final es un código de HTML con CSS donde se puede visualizar claramente los diferentes tipos de caracteres identificados. Lo que más se me dificultó en el trabajo fue añadir las nuevas funciones y estados al programa de los tokens ya que la matriz de transiciones no me había quedado muy claro pero logré entenderme con mis compañeros y se logró completarlo. Lo más fácil fue generar el código del output final aunque no tenía muy claro como hacerlo desde el código de python, el código en sí para colorear y presentarlo no generó dificultades. La complejidad de nuestros programas son de O(n). Cuando no había errores, se escriben los tokens válidos en el HTML con sus etiquetas y clases. Si sale un error, se muestra un mensaje de error y se cierran las etiquetas. Al terminar se cierran todas las etiquetas para garantizar un formato html para situaciones con y sin errores. Finalmente todo funcionó como se esperaba.

Autómata


dig
op ( ) ; ,
100
dig
dig
delimitador
101
operación
102
(
103
)
104
END
105
,
106
=
107
id
letra, dig
delimitador
108
delimitador
110
bool
200
error