**Trabajo Práctico**

**Algoritmos y Programación I**

Grupo: Ballenato

Integrantes:

* Camila Bartocci
* Elián Foppiano
* Santiago Marczewski
* Gastón Proz
* Jean Paul Yatim Este

Corrector: Lic. Gustavo Bianchi

Fecha de entrega: 23/07

**Hipótesis de trabajo**

La aplicación a analizar debe respetar una cierta estructura, a fin de que el analizador de texto funcione correctamente.

1. El código solo puede contener elementos de la programación estructurada, tales como las estructuras secuenciales, de selección e iterativas, agrupadas en funciones.
2. El módulo principal debe invocar a una única función, que será la función principal, en la cual se invocarán al resto de las funciones, estén definidas en el mismo módulo o no. Si no fuera así, la funcionalidad 4 (árbol de invocación) no tendría una función por la cual empezar el recorrido de invocaciones.
3. El programa no puede contener funciones de igual nombre definidas en módulos diferentes.

**Sobre las reuniones**

**Reunión 1.** Previo a esta reunión, cada integrante leyó detenidamente el enunciado del trabajo y se creó un repositorio del grupo en GitHub. Durante el encuentro, se discutió sobre las consignas del trabajo, se aclararon algunas dudas y se establecieron las hipótesis de este. También se trabajó sobre la funcionalidad 0, al analizar un esqueleto de dicha función. De esta reunión, surgieron las preguntas que fueron plasmadas en el foro, posteriormente respondidas en clase, y también una primera aproximación a cómo podrían encararse las funcionalidades, dichas ideas luego detalladas en un documento que contenía las propuestas de pautas para realizar el trabajo.

**Reunión 2.** Para esta reunión, ya teníamos divididos los enunciados por integrante del grupo y ya habíamos empezado a trabajar sobre los puntos asignados. Se decidió que cada uno se encargaría de una funcionalidad y luego se haría un comentario general al grupo de cómo se llevó a cabo el trabajo individual, usando como guía las propuestas que resultaron de la reunión anterior. Además, surgieron algunas dudas más que fueron plasmadas en el foro. Los avances del proyecto se fueron subiendo al repositorio de GitHub a lo largo de la semana.

**Reunión 3.** El objetivo de esta reunión fue mostrar los avances de la semana. Teníamos las funcionalidades ya muy avanzadas, tratamos de resolver problemas puntuales que se iban encontrando en los códigos de cada integrante, y ya empezamos a pensar en qué se podía mejorar en algunas funciones. También se trató el tema de la integración del proyecto y la posibilidad de reutilizar código, creando un módulo que contenga funciones repetidas en los programas, por ejemplo, la función que se encarga de leer una línea del csv, presente en todos los módulos de los integrantes.

**Reunión 4.** En esta reunión, se terminaron de corregir cosas puntuales de los módulos hechos por cada integrante. Se realizó una prueba general de todo el programa con los archivos app\_matematica.py, lib\_matematica.py y algunas de las mismas funcionalidades programadas para el trabajo. A su vez, se trataron temas de estética en cuanto a las tablas impresas y los archivos generados en cada funcionalidad. Se creó el módulo *universales* que contiene funciones comunes a distintos módulos: leer\_lineas\_csv, obtener\_lista\_funciones y obtener\_comentario\_multilinea. Por último, se realizó una limpieza del repositorio en GitHub.

**Reunión 5.** En este encuentro, se hicieron pruebas sobre el proyecto con los propios archivos que forman parte del programa. También se trabajó sobre la documentación del .docx y descripciones (comentarios) en el código de cada módulo. Por último, se arreglaron detalles de algunas funciones puntuales.

El funcionamiento y las particularidades de cada funcionalidad y los cambios significativos que se tuvieron que realizar en cada módulo, en cuanto a su estructura, se detallan a continuación.

**Documentación de los módulos**

**Módulos auxiliares**

**Exp\_reg**

Este módulo reúne diversas funciones relacionadas con expresiones regulares y el manejo de cadenas. Surgió la idea de crearlo debido a la complejidad de implementar ciertas funciones con estructuras tradicionales. Con la ayuda de las expresiones regulares, el código se simplificó y la legibilidad aumentó.

**Universal**

Al combinar cada una de las funcionalidades en el programa principal, detectamos que algunas funciones definidas en distintos módulos realizaban la misma tarea. Por lo tanto, creamos un módulo en el cual estuvieran definidas para que se pueda acceder a ellas de forma práctica.

**‘Funcionalidad 0’ – preprocesamiento de archivos**

Los módulos de la “funcionalidad 0” corresponden a todo lo relacionado con el preprocesamiento de los archivos originales para la creación de los .csv. Aquellos módulos que dicen “descontinuado”, son módulos que permanecieron en la aplicación durante un tiempo significativo, y que luego fueron eliminados al no ser necesarios, gracias a las mejoras en los algoritmos.

1. **Ordenar**

Es el módulo que se encarga de realizar un análisis preliminar de los archivos recibidos. Dada una lista de programas, los manipula de a uno, cargando todas las funciones del mismo en memoria, ordenándolas alfabéticamente por el nombre, y guardándolas en la carpeta “funciones” para luego ser utilizadas por el siguiente módulo. En el programa principal también se detecta cuál es la función que se invoca por fuera de cualquier bloque de código (función principal), y al momento de copiarla le agrega un carácter especial por delante del nombre (marcador principal), que en este caso es el símbolo “$”. Para facilitar el manejo posterior de las funciones, al momento de copiarlas se reemplazan todas las comillas simples por comillas dobles, las tabulaciones por 4 espacios en blanco, y se omiten las líneas en blanco.

1. **Generar\_archivos\_csv**

Toma los archivos generados por el módulo anterior y extrae la información para crear los archivos fuente\_unico.csv y comentarios.csv. Para guardarlas de manera ordenada alfabéticamente, se utiliza una modificación del algoritmo de mezcla, capaz de manejar una cantidad variable de archivos. El código de cada función debe ser tratado de forma diferente dependiendo de qué archivo está generando.

Al crear fuente\_unico.csv, primero se guarda el nombre de la función y los parámetros, que se encuentran en la firma de la función. Luego se guarda el nombre del módulo en el cual está definida. Después se saltea el comentario inicial, si es que existe, y comienza la lectura y guardado de las instrucciones del código. Para ello se recorre el archivo hasta que se encuentre la siguiente firma de función, distinguiendo el código entre instrucciones, comentarios de línea (empiezan con #), y comentarios multilínea. Si se trata de una instrucción, la guarda eliminando, si existe, el comentario que le sigue en la misma línea. Si es un comentario de línea, no se hace nada. Si es un comentario multilínea, se recorre completamente y no se guarda. En cada ciclo se aumenta una variable que guarda la posición en la que se encontró cada línea o bloque de comentario.

Al crear comentarios.csv, primero se guarda el nombre de la función. Si luego de esa línea empieza un comentario multilínea, se trata del comentario inicial de la función, y se analiza para extraer el nombre del autor y la descripción de ayuda. Si no se encuentran dichos datos o no hay un comentario inicial, la función se guarda con autor “Desconocido” y el campo de descripción de ayuda se guarda en blanco. Luego se guardan los comentarios adicionales que puedan existir en la función. Al igual que en fuente\_unico.csv, se recorre el archivo mientras no se encuentre una nueva firma de función. Si se encuentra un comentario multilínea, se recorre completamente y se guarda en un único campo. Si se encuentra un comentario de línea, también se guarda. Si no es ninguno de los casos anteriores, es porque la línea es una instrucción, pero que puede contener un comentario, por lo que se guarda también si lo tuviera. Al igual que antes, se mantiene registro de la posición en la que se encontró cada línea.

Cada línea adicional de los .csv están precedidas por un marcador de línea, que da idea de la posición en la que se encuentra cada cosa. El marcador es de la forma “/*x*/” donde *x* es un número, empezando desde el 0 (primera instrucción o bloque de comentarios posterior al comentario inicial). Adicionalmente, en las primeras versiones del programa se guardaba un marcador que indicaba el nivel de indentación al que se producía cada línea, pero luego fue removido ya que no era necesario.

Para evitar problemas al leer los archivos .csv, al guardar un salto de línea dentro de un comentario multilínea, se reemplaza el caracter \n por el marcador “/n/”; y las comas se reemplazan por el marcador “/c/”.

1. **Manejo\_imports (descontinuado)**

Se encargaba de crear un archivo “imports.csv” en el cual se guardaban los nombres de los programas y los módulos que importaba cada uno. Fue descontinuado, ya que la única funcionalidad que la utilizaba (arbol\_invocacion) fue modificada para buscar las funciones de manera indistinta.

1. **Lista\_funciones (descontinuado)**

Generaba el archivo “funciones\_por\_modulo.csv” que guardaba los nombres de los programas y las funciones que definidas en el mismo. Fue descontinuado por el mismo motivo que manejo\_imports.

1. **Panel general de funciones**

Este módulo se encarga de mostrar por pantalla una tabla que contiene, para cada función, la siguiente información: nombre de la función y módulo al que pertenece, cantidad de parámetros, cantidad de líneas de código, cantidad de invocaciones, cantidad de return, if/elif, for, while, break, exit, cantidad de líneas de comentarios, indicador de descripción y autor de la función. Además, genera el archivo panel\_general.csv, en el cual la primera línea contiene las etiquetas de lo que contiene cada campo. Se almacenan los datos pedidos en un diccionario, el cual tiene la siguiente forma:

{

“func\_1”:{ “nombre.modulo”:”func\_1.modulo”, “parámetros”:no,

“líneas”:n1, “invocaciones”:n2, “returns”:n3, “ifs”:n4,

“fors”:n5, “whiles”:n6, “breaks”:n7, “exits”:n8,

“comentarios”:n9, “descripcion”: “si”/”no”,

“autor”:”nombre apellido”},

“func\_2”:{ “nombre.modulo”:”func\_2.modulo”, …},

“func\_n”:{ … }

}

La función nombre\_funcion, es con la cual se empiezan a almacenar los pares “funcion”:{datos} y, a su vez, le agrega el primer dato al diccionario. Las demás funciones se encargan de agregar los otros datos a cada diccionario de valor de cada clave, exceptuando: formato\_tabla, que da formato a la tabla que se imprime por pantalla, genera\_dic, que abre fuente\_unico.csv y comentarios.csv y genera el diccionario haciendo los llamados a las funciones, genera\_panel\_csv, que se encarga de crear panel\_general.csv en la carpeta funcionalidades, y funcionalidad\_panel, la cual articula formato\_tabla y genera\_panel\_csv para ejecutar la funcionalidad 1.

A cada función se le pasa por parámetro el diccionario donde se almacenarán los datos y el archivo del que se extraerá la información (fuente\_unico.csv o comentarios.csv), exceptuando las funciones que no almacenan datos en el diccionario

Cada función, exceptuando las que no se encargan de almacenar datos en el diccionario, toma el nombre de la función a analizar y lo almacena en la variable *funcion*, para luego poder agregar el nuevo dato de la función correspondiente al diccionario, y trabajar con éste usando índices y haciendo referencia al nombre de la función. También, siempre se realiza un seek(0) sobre el archivo, para asegurarse de que el puntero esté señalando siempre al principio del archivo y así evitar errores.

*Nota sobre la impresión de la tabla en pantalla: en caso de que el campo de “nombre.modulo” o el de “autor” sean muy largos, será necesario hacer un zoom out para ver la tabla completa.*

En un principio, se había realizado el módulo a través de listas. Cada ítem de la funcionalidad 1 se almacenaba en listas. Se tenían así 13 listas, sin referencia alguna a qué significaban los elementos de dichas listas. Se trabajaba todo con índices, por ejemplo, para la funcionalidad de la tabla: se sabía, por la naturaleza de las listas, que el primer elemento contenía la cadena con el nombre y el módulo, el segundo elemento era un valor numérico que representaba cantidad de parámetros, el tercer elemento era otro valor numérico que representaba cantidad de líneas, etc.

Una vez que el módulo ya estaba casi finalizado, se optó por modificar el código y trabajar con un diccionario en vez de listas aisladas. Sería más eficiente a la hora de almacenar los datos, ya que quedan mucho más ordenados y es más claro saber qué significa cada valor, además de que se facilita la lectura del código y la tarea de buscar algún valor, pues estaría explícito de qué se trata por la naturaleza del diccionario. Un ejemplo sería dic[“func\_1”][“parametros”] lo cual es sin duda más entendible que lista\_parametros[0]. Además, con la idea descartada de las listas, para la función que da formato de tabla, se habían implementado todavía más listas, las cuales almacenaban los valores de cada una de las 13 listas, por cada función que existía en el programa analizado. Esto, con el diccionario se facilitó muchísimo, pues ahora sólo había que recorrer el diccionario y tomar cada elemento (cada función) y utilizar la sintaxis antes mencionada.

Este módulo importa el módulo *universales*, para utilizar la función leer\_lineas\_csv. Esto reemplaza la función leer\_archivo, que se utilizaba en este módulo para leer cada línea de los csv.

Este módulo también importa el módulo exp\_reg, para hacer uso de la función contar\_invocaciones, que es utilizada en las funciones cant\_invocaciones y cant\_estructuras.

Por un lado, antes de implementar la optimización del código con diccionarios, se había llegado a una solución para la actual función cant\_invocaciones, que incluía dos funciones en vez de una sola. Una de las funciones, tomaba la lista que correspondía a nombre.modulo, y se quedaba sólo con el nombre de la función, almacenándolo en una lista que se devolvía. Luego la segunda función, a través de ciclos while y for y varios ifs (todos estos anidados), se encargaba de guardar cada invocación en un diccionario, que tenía como clave la función y como valor la cantidad de llamados. Esto no era eficiente, ya que se tenían muchos niveles de anidación, mezcla de for, while e ifs, y encima se estaban generando una lista y un diccionario extra, además de que era una función que tenía más de 30 líneas de código. En la etapa de optimización, se logró reducir el problema a una función, importando contar\_invocaciones, lo cual mejoró mucho el código, ya que ahora no hay anidaciones de dichas estructuras y la función es mucho más corta.

Por otra parte, antes se había llegado a una solución para cant\_estructuras, en la cual se hacía uso de expresiones regulares y se repetía mucho código, además de que la función tenía 30 líneas de código. Entonces, en la etapa de optimización del módulo, se optó por hacer una modificación de la funcionalidad contar\_invocaciones del módulo exp\_reg, para que permita también contar estructuras y poder así reutilizar ese código. Así se logró reducir la cantidad de líneas de cant\_estructuras, invocando dicha función.

1. **Consulta de funciones**
2. **Analizador de reutilización de código**

Para esta funcionalidad, se tenía que devolver una tabla que mostrara las invocaciones que hace cada función y qué funciones son invocadas por cuales. Para eso, la idea inicial fue crear una lista con los nombres de todas las funciones en el programa, sacándolos del primer campo de cada fila en fuente\_unico.csv, de modo tal que no haría falta volver a recorrer el archivo para encontrar cada invocación.

Una vez que se tenía la lista con las funciones, se podía volver a recorrer fuente\_unico.csv pero esta vez chequeando, para cada función en la lista, si se encontraba invocada en sus líneas de código (a partir del cuarto campo de cada línea). Se había inicialmente interpretado que una función era invocada cuando aparecía como "nombre\_de\_funcion(". Después se descubrió un bug porque habían funciones cuyos nombres hacían parecer que eran invocadas cuando en realidad no era el caso (por ejemplo, "función(" y "otra\_función(" las contaba como la misma función). Entonces se utilizó una expresión regular que chequea que antes de "nombre\_de\_funcion(" haya un carácter de separación de palabras. Se creó un diccionario con todas las funciones como claves. Cada clave tiene como valor otro diccionario. Este segundo diccionario contiene las funciones que invoca como claves y cuántas veces la invoca como su valor. De este modo, Si la función "A" invoca a "B" 3 veces, "B" invoca a "C" 2 veces y a "D" 1 vez y ni "C" ni "D" invocan a nadie, el diccionario queda como: {A:{B:3}, B:{C:2, D:1}, C:{}, D:{}}.

A continuación, había que crear la tabla que el programa iba a mostrar. Como además de mostrarla había que también crear un archivo "analizador.txt" que contenga la misma tabla, primero se creó la tabla en el "analizador.txt" para después imprimir directamente del archivo cuando tocara mostrarla.

Para crear la tabla en el archivo se tenía que escribir fila por fila. Para separar filas se utilizó "-" y para separar columnas, "|". Para la primer columna, en la primer fila, se muestra "FUNCIONES", debajo, cada fila contiene el nombre de una función con un numero identificador (del 1 al número de funciones en el programa). Para las siguientes columnas, en la primer fila de cada una están los números identificadores. Por lo tanto, el numero de columnas y el de filas es igual al número de funciones en el programa. Como el formato de cada fila depende de las invocaciones de cada función, se creó un formato especial para las filas, donde: si la función de la fila invoca a la de la columna, muestra el número de veces que la invoca (también ese numero lo suma a un diccionario como valor del total de veces que la función de la columna es invocada); si la función de la fila es invocada por la de la columna, muestra "x"; y, si no ocurre ninguna invocación entre las funciones, deja un espacio en blanco. Para escribirlo todo en la tabla se utilizó un "for" que escribe en el archivo de texto, para cada función, el formato del nombre con su identificador, seguido por el formato especial de la fila. Al final, como última fila de la tabla, se agregó, en la primer columna el título "Total invocaciones", y en las siguientes columnas, los valores totales en el diccionario del total de invocaciones.

Finalmente, se creó la función que, cuando es llamada, imprime la tabla dentro del archivo "analizador.txt".

1. **Árbol\_invocacion**

La estructura principal utilizada es un diccionario cuyas claves son los nombres de todas las funciones del programa, y el valor asociado es una lista con los nombres de todas las funciones que invoca. Adicionalmente, se crea un diccionario que guarda la cantidad de instrucciones de cada función. La función principal del módulo toma estos diccionarios junto con el nombre de la función principal (identificada en los .csv con el marcador principal), y comienza a imprimirlas de manera recursiva. Por cada función que se imprime, se aumenta un acumulador que lleva registro del espacio que ocupa cada función, para poder imprimirlas con el nivel de indentacion adecuado. La primera rama se imprime sin dicha indentación, pues es la que va en la misma línea que la función que la invoca. Cuando se termina una rama (se llega a una función que no invoca a ninguna otra), se imprime un salto de linea y las siguientes funciones se imprimen con la indentación adecuada.

Uno de los inconvenientes que surge al aplicar esta solución, es que las funciones recursivas se imprimirían de manera indefinida en el árbol. Y puesto que este mismo módulo implementa una, no era un problema que se pudiera ignorar. Para solucionarlo, antes de comenzar a imprimir se detectan aquellas funciones que se llaman a sí mismas y se cambia el nombre de la función en la lista de invocaciones por “nombre\_funcion (Recursivo)”. Luego se crea un nuevo registro en el diccionario con ese nombre, que no invoque a ninguna otra. De esta manera, solo se imprime un “nivel” de recursividad, y luego se corta.

Otro de los inconvenientes era identificar a la funcion principal dentro de los .csv, para que el árbol comenzara desde ella. Surgieron dos opciones factibles para solucionarlo. La primera era detectar la funcion principal al momento de crear los .csv y marcarla de alguna manera para encontrarla posteriormente con ese marcador. La segunda era buscar la función que no es invocada por ninguna otra. Esta última solución tiene el inconveniente de que, si existen funciones que no poseen una implementación directa en la aplicación, no se podría detectar la funcion principal correctamente (el algoritmo no distinguiría entre la principal y una sin implementación). Debido a esto, se optó por la primera solución.

Para buscar las funciones e invocaciones, el programa hacía uso de los módulos manejo\_imports y lista\_funciones, con el fin de buscar únicamente las funciones definidas en los módulos que se importaban. Pero la mejora en la eficiencia no era apreciable, por lo que se optó por buscar todas las funciones en todos los módulos, con lo cual el código se simplificaba y los módulos que generaban los .csv extra fueron descontinuados.

**5. Informe por desarrollador.**

La estructura utilizada es la de un diccionario principal que tiene como clave los autores y dentro una lista con la firma de las funciones como primer campo, y la cantidad de líneas de dicha función como segundo campo, toda esa información proviene de fuente\_unico.csv y comentarios.csv. Todos los cálculos realizados por las otras funciones extraen la información del diccionario principal sin necesidad de seguir leyendo los archivos csv. La estructura de los cálculos que devuelve cada función está compuesta por diccionarios con autores como claves, y con la información correspondiente de lo calculado como valor.

Esto está hecho de esta forma para que a la hora de iterar sobre cada información que se necesite de cada autor se pueda hacer de una forma más fácil e intuitiva.

-Diccionario por autor:

La estructura del diccionario es de la siguiente forma:

{“Autor”:[(funcion1, líneas), (funcion2, lineas), …], “Autor2”: [(función, líneas), (función, lineas), …]}