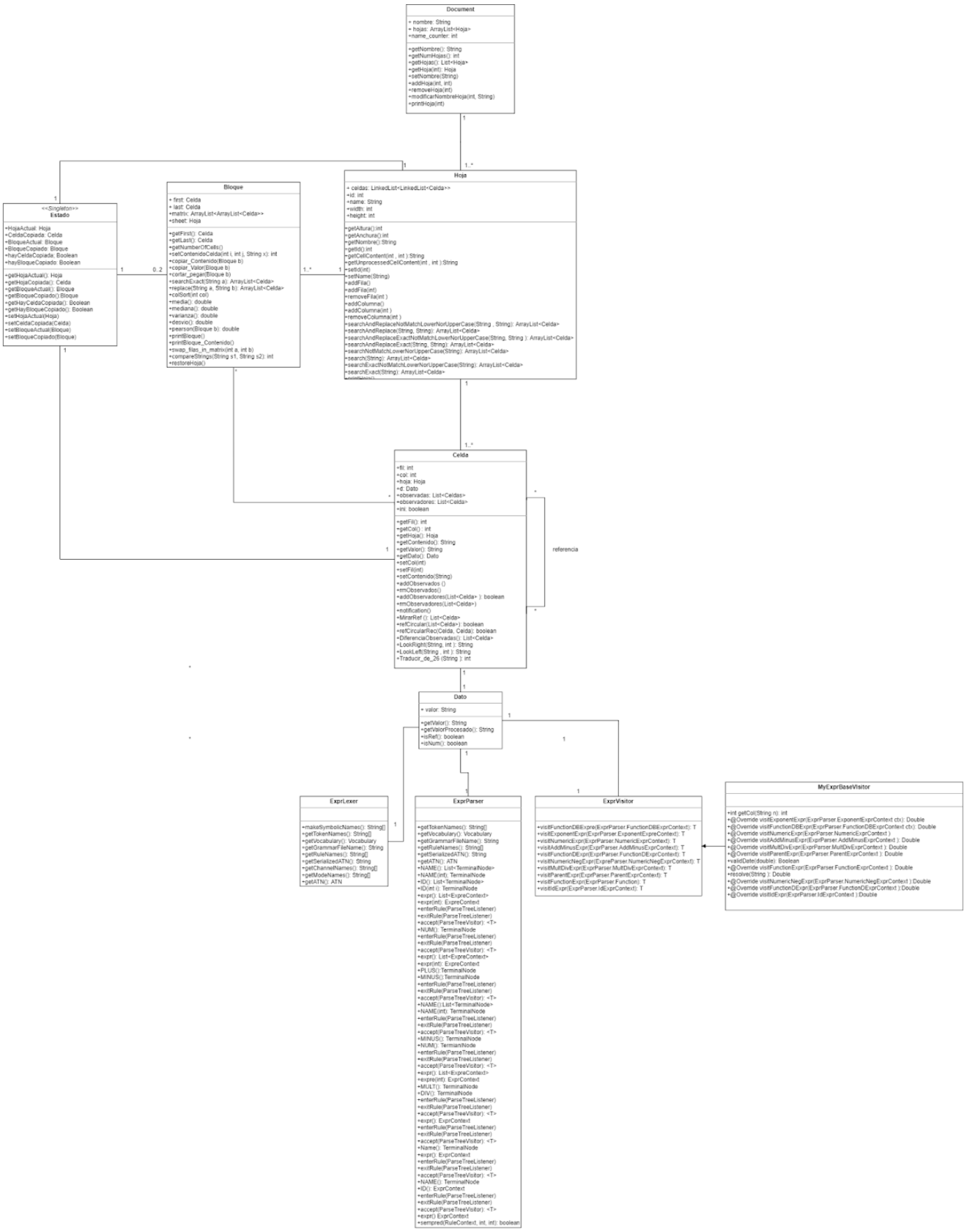
PROP
PRIMERA ENTREGA
Descripciones de clases, algoritmos y estructuras de datos

Javier Abella Nieto
Alejandro Durán Saborido
Joaquín Faraone Prieto
Lluís Pujalte Feliu De Cabrera

24/04/2022
2021-22 Q2

Índice

1. Diagrama de clases



2. Relación de clases por cada miembro del equipo

2.1 Clases del diagrama de clases

- Documento: Alejandro Durán Saborido
- Hoja: Alejandro Durán Saborido
- Celda: Javier Abella Nieto
- Bloque: Joaquín Faraone Prieto
- Dato (y sus derivados): Lluís Pujalte Feliu De Cabrera
- Estado: Alejandro Durán Saborido y Lluís Pujalte Feliu De Cabrera

2.2 Otras clases

- Main: Alejandro Durán Saborido y Joaquín Faraone Prieto
- CtrlDominio: Alejandro Durán Saborido

3. Descripción de las Clases

3.1 Documento

Autor: Alejandro Durán Saborido

La clase Documento representa un documento de tipo hoja de cálculo. Cada documento se identifica por su nombre (*string*) y contiene un conjunto de objetos de la clase Hoja que representan las hojas del documento. Este conjunto se guarda en un *ArrayList* llamado hojas.

La creadora de la clase recibe dos enteros, que representan el número de filas y columnas de la primera hoja del documento pues un documento debe tener siempre al menos una hoja, y un *string* que será el nombre. Solo existe una instancia de documento activa a la vez, si se desea crear otra, se deberá cerrar el documento actual primero.

Las hojas dentro del documento están indexadas empezando por 1 (aunque internamente empiezan por 0).

Funciones

- ***addHoja(int height, int width)***: añade una hoja de tamaño *height* x *width* al final del documento.

- ***removeHoja(int index)***: elimina la hoja en el índice especificado con *index*.

- ***modificarNombreHoja(int index, String nombre)***: modifica el nombre de la hoja en el índice especificado con *index* a *nombre.*

- ***printHoja(int id)***: imprime por terminal el contenido de la hoja especificada por *id.*

3.2 Hoja

Autor: Alejandro Durán Saborido

La clase Hoja representa una página del documento de plantilla de cálculo. Cada hoja se identifica por un int *id* que representa su posición dentro del documento. Además, tiene de atributos un String *name* que representa el nombre de la hoja, dos enteros *height* y *width* que representan el número de filas y de columnas de la página respectivamente y una matriz representada mediante una *LinkedList* de *LinkedList* de objetos de tipo Celda.

La estructura *LinkedList* almacena sus elementos en "contenedores". La lista tiene un enlace al primer contenedor y cada contenedor tiene un enlace al siguiente contenedor de la lista. Para agregar un elemento a la lista, el elemento se coloca en un nuevo contenedor y ese contenedor se vincula a uno de los otros contenedores en la lista, por lo que aunque no es muy eficiente para almacenar información, sí lo es para manipularla, lo cual es justamente lo que necesita para representar una hoja de cálculo y su contenido el cual debe ser fácilmente manipulable.

También se consideró utilizar un *ArrayList* pero estos a pesar de ser más eficientes para almacenar datos, no lo son tanto para manipularlos, por lo que finalmente se optó por la alternativa de *LinkedList*.

Cabe destacar que las coordenadas de las celdas dentro de la hoja comienzan en el (1,1) (aunque internamente empiezan en (0,0)) empiezan arriba a la izquierda y van de izquierda a derecha y de arriba a abajo.

Funciones

- ***getCellContent(int x, int y)***: esta función recibe la fila (*x*) y la columna (*y*) de una celda y devuelve su contenido procesado por la clase Dato.
- ***getUnprocessedCellContent(int x, int y)***: esta función recibe la fila (*x*) y la columna (*y*) de una celda y devuelve su contenido sin haber sido procesado por la clase Dato.
- ***setContenidoCelda(int x, int y, String newData)***: fija como contenido el String *newData* en la celda identificada por la fila *x* y la columna *y*.
- ***addFila()***: añade una nueva fila lo más abajo posible.
- ***addFila(int index)***: añade una nueva fila en la posición especificada por *index*, mueve una posición hacia abajo todas las filas que estén por debajo de esa posición.
- ***removeFila(int index)***: elimina la fila en la posición identificada por *index*.
- ***addColumna()***: añade una nueva columna lo más a la derecha posible.

- ***addColumna(int index)***: añade una nueva columna en la posición especificada por *index*, mueve una posición hacia la derecha todas las filas que estén a la derecha de esa posición.

- ***removeColumna(int index)***: elimina la columna en la posición identificada por *index*.

- **Funciones de búsqueda y reemplazo**:
    - ***searchAndReplaceNotMatchLowerNorUpperCase(String search, String replace)***: busca en el contenido procesado de todas las celdas de la hoja el *substring search* (sin coincidir mayúsculas y minúsculas) y lo reemplaza por el *string replace*. Devuelve un *ArrayList* con las celdas modificadas.
    - ***searchAndReplace(String search, String replace)***: busca en el contenido procesado de todas las celdas de la hoja el *substring search* (coincidiendo mayúsculas y minúsculas) y lo reemplaza por el *string replace*. Devuelve un *ArrayList* con las celdas modificadas.
    - ***searchAndReplaceExactNotMatchLowerNorUpperCase(String search, String replace)***: busca las celdas de la hoja cuyo contenido procesado sea igual al *string search* (sin coincidir mayúsculas y minúsculas) y lo reemplaza por el *string replace*. Devuelve un *ArrayList* con las celdas modificadas.
    - ***searchAndReplaceExact(String search, String replace)***: busca las celdas de la hoja cuyo contenido procesado sea igual al *string search* (coincidiendo mayúsculas y minúsculas) y lo reemplaza por el *string replace*. Devuelve un *ArrayList* con las celdas modificadas.
    - ***searchNotMatchLowerNorUpperCase(String search, String replace)***: busca en el contenido procesado de todas las celdas de la hoja el *substring search* (sin coincidir mayúsculas y minúsculas). Devuelve un *ArrayList* con las celdas que lo contengan.
    - ***search(String search, String replace)***: busca en el contenido procesado de todas las celdas de la hoja el *substring search* (coincidiendo mayúsculas y minúsculas). Devuelve un *ArrayList* con las celdas que lo contengan.
    - ***searchExactNotMatchLowerNorUpperCase(String search, String replace)***: busca las celdas de la hoja cuyo contenido procesado sea igual al *string search* (sin coincidir mayúsculas y minúsculas). Devuelve un *ArrayList* con las celdas que lo contengan.
    - ***searchExact(String search, String replace)***: busca las celdas de la hoja cuyo contenido procesado sea igual al *string search* (coincidiendo mayúsculas y minúsculas). Devuelve un *ArrayList* con las celdas que lo contengan.

- ***printHoja()***: imprime la hoja por pantalla, mostrando el contenido de todas las celdas en forma de matriz.

3.3 Celda

Autor: Javier Abella Nieto

La clase Celda representa una celda dentro de una hoja de un documento. Esta se identifica por sus tres coordenadas, fila, columna y hoja.

Celda incorpora un puntero a la hoja que pertenece para poder acceder a las celdas necesarias para el patrón observador, explicado más abajo.

Celda incorpora un booleano inicializado llamado ini para así poder saber si la celda está vacía o no para que la clase Dato pudiera resolverse de manera más eficiente.

La clase celda incorpora un patrón observador* implementado mediante el uso de dos listas de celdas, la lista “observadas” nos indica cuales son las celdas que son referenciadas por la celda actual, mientras que la lista “observadores” nos indica cuales son las celdas que están referenciando a la celda actual. Cuando una celda que esté referenciada por otra celda experimenta un cambio de estado (cambio en el contenido o en su valor) se envía una señal a todas sus celdas observadores para que éstas puedan mantenerse actualizadas recalculando su nuevo valor utilizando el nuevo valor de la celda observada.

Patrón observador: Patrón de diseño que nos permite actuar ante cambios de estado de un elemento observado por un observador.

Funciones

- ***setContenido*:** Cambia el contenido de una celda y activa el patrón observador en caso que se tenga que modificar las listas o notificar una actualización. En caso de error por autorreferencia o referencia circular se deja el contenido de la celda tal y como estaba antes de llamar a esta función.

- ***MirarRef:*** Devuelve una lista de las celdas que están siendo referenciadas por la actual y avisa si hay una autorreferencia.

- ***addObservados:*** Añade a la lista de “observadores” todas las celdas nuevas que están siendo referenciadas por la celda actual.

- ***rmObservados:*** Elimina de la lista de “observados” todas las celdas que ya no están siendo referenciadas por la celda actual.

- ***addObservadores:*** Añade la celda actual a la lista de “observadores” de las celdas que referencia.

- ***rmObservadores:*** Elimina la celda actual de la lista de “observadores” de las celdas que referenciaba.

- ***RefCircular:*** Esta función devuelve verdadero si se detecta una referencia circular o falso si no se detecta.

- ***RefCircularRec:*** Esta función se recorre de manera recursiva todas las referencias que dependen de otras referencias para detectar o no una referencia circular.

- ***DiferenciaObservadas:***: Esta función devuelve la diferencia entre dos listas de celdas, la lista “observadas” - la lista sacada de la función ***MirarRef.***

- ***Notification***: Se utiliza cuando una celda ha experimentado un cambio de contenido y debe avisar a sus observadores para que tengan su valor actualizado.

3.4 Bloque

Autor: Joaquin Faraone

Definimos un bloque de celdas como un subgrupo **n*x*m** de celdas. La clase Bloque representa exactamente eso. Las celdas de un bloque cumplen con pertenecer todas a una única hoja de ser adyacentes entre ellas.Otra forma de definirlo puede ser; todas las celdas comprendidas entre un fila i y la fila i+n y entre una columna j y la columna j+m.

La clase celda fue concebida con 4 atributos. La primera celda (aquella en la esquina superior izquierda), su última celda (aquella en la esquina inferior derecha), la hoja a la que pertenecen todas las celdas del bloque y un conjunto (ArrayList) de celdas el cual la constituyen.

Vale la pena destacar que los atributos “primera celda”, “última celda” y Hoja son redundantes dado a que esta información puede ser obtenida directamente del conjunto ordenado de las celdas (el 4to atributo de la clase). Sin embargo, se optó por mantenerlos dado que los tres constituyen la clave maestra de la clase. Al dejarlos, estos atributos han contribuido a controlar errores y evitar inconsistencias.

Para el atributo del subconjunto de celdas que constituyen el bloque se decidió trabajar con una matriz, un arreglo de arreglos de celdas. Se utilizó la clase ArrayList de java en ambas dimensiones. Se eligió esta estructura en base a la naturaleza del trabajo destinado al uso de un bloque. Al seleccionar un bloque un usuario busca un trabajo puntual, fino, y de menor volumen en comparación con el de trabajar con toda una hoja. Por eso esta estructura, que ofrece rápido acceso a sus elementos, resulta atractiva. Se tomó la decisión objetiva sobre una presunción subjetiva del trabajo del usuario.

En la práctica, y al implementar las funcionalidades de la misma, no se saca mayores provecho de las ventajas del arreglo. Muchas funciones hacen recorridos completos de todas las celdas con lo cual otras estructuras (como por ejemplo una linkedList al igual que Hoja) también hubiesen sido útiles. Sin embargo, funciones como la de ordenar por columna son más eficientes debido a la estructura seleccionada.

La función creadora de la clase debe recibir dos celdas correspondientes a dos esquinas (siempre cruzadas) del bloque con el cual se desea trabajar. Una vez con el bloque creado, las funciones de la clase pueden ser divididas de dos maneras. Funciones informativas y funciones modificadoras. El primer grupo ofrece información sobre los valores comprendidos dentro del bloque. Estas pueden ser usadas directamente por el usuario como internamente mediante una referencia dentro de una celda. El segundo grupo, las funciones modificadoras ejecutan un cierto cambio dentro de la hoja que contiene el bloque.

A seguir debemos hacer distinción entre los terminos “contenido” y “valor”. Por “contenido de/en una celda” nos referiremos a exactamente lo introducido por el usuario en una celda. Mientras que al hacer uso de “valor de/en una celda” estaremos refiriendonos a la interpretasion (hecha por el software) del contenido de la celda.

Funciones

Funciones modificadoras

- ***Copiar_Contenido:*** Esta función recibe como parámetro un segundo bloque y su labor es copiar el contenido del bloque al segundo bloque. Es decir, una vez ejecutada la función el contenido de la celda ixj del segundo bloque es idéntico al contenido de la celda ixj del bloque original. El bloque recibido como parámetro debe poseer las mismas dimensiones que el bloque original. *Ver nota 1 y 2*

- ***Copiar_Valor:*** Función análoga a “copiar contenido”; en la cual el segundo bloque (pasado por parámetro) recibe los valores procesados del bloque. Las referencias, y fórmulas se pierden con esta función dejando apenas. Es decir, una vez ejecutada la función el contenido de la celda ixj del segundo bloque es idéntico a lo valor del contenido de la celda ixj del bloque original. El bloque recibido como parámetro debe poseer las mismas dimensiones que el bloque original. *Ver nota 1 y 2*

- ***Cortar_Pegar:*** Función análoga a “copiar contenido”. El contenido del bloque es copiado a un segundo bloque de iguales dimensiones (pasado como parámetro). A diferencia con “copiar contenido”, en esta función, las celdas del bloque original quedan con contenido vacío. *Ver nota 1 y 2*

- **Replace:** La función recibe dos parámetros de tipo String. Análogamente a la función “search exact” busca el primer String recibido. Al encontrarlo en el contenido de una celda, el contenido de esta misma celda es sustituido exactamente por el segundo String de entrada. La función devuelve el subconjunto de celdas (en un ArrayList) en las cuales se haya efectuado la operación de intercambio.

- **ColSort:** La función recibe como parámetro de entrada una variable de tipo entero (int). El entero (i) debe ser menor que la cantidad de columnas en el bloque. Una vez ejecutada la función, las filas del bloque cambian de posicion entre si de tal manera que los valores de las celdas en la columna “i” estan ordenados crecientemente. La distribución de las celdas en una fila en particular no sufre alteraciones. *Ver nota 3*

Funciones informativas

- **Search Exact:** La función recibe un parámetro de entrada tipo String. Busca dentro de las celdas que conforman el bloque y devuelve un subconjunto de celdas (en un ArrayList) todas las celdas cuyo Contenido o valor sea idéntico al String recibido.

- **Funciones estadísticas:** El conjunto de las siguientes funciones considera que todos los valores dentro del bloque son de tipo numérico.
    - Media: Devuelve el promedio de los valores contenidos en todas las celdas comprendidas en el bloque.
    - Mediana: Devuelve la mediana de los valores de todas las celdas comprendidas en el bloque.

- Varianza: Devuelve la varianza de los valores de todas las celdas comprendidas en el bloque.
- Desvío (estándar): Devuelve el desvío estándar (estándar) de los valores de todas las celdas comprendidas en el bloque.
- Covarianza: La función recibe como parámetro de entrada otro bloque. Devuelve la covarianza entre los valores de las celdas de un bloque en relación con el otro. El bloque de entrada debe ser de las mismas dimensiones que el bloque original.
- Pearson (coeficiente de): La función recibe como parámetro de entrada otro bloque. Devuelve el coeficiente de Pearson entre los valores de las celdas de un bloque en relación con el otro. El bloque de entrada debe ser de las mismas dimensiones que el bloque original.

Nota 1: En esta primera version del software se decidio darle más responsabilidad al usuario de la habitual en las funcionalidades de copiar/cortar y pegar. El usuario debe indicar adecuadamente el bloque destino al momento de realizar el comando. Otro funcionamiento fue previsto aunque no implementado. La clase “estado” prevee el atributo Bloquecopiado para separar las etapas de copiar un bloque y pegarlo posteriormente.

Nota 2: Como medida global del diseño del programa, se tomó la desicion conjunta de que una intancia de bloque no podra trasender fuera de la hoja que contiene sus celdas. Es decir, un bloque no puede ser referenciado desde celdas de otra hoja y un bloque no puede modificar celdas de una hoja a la cual no pertenece. En la practica, para respetar esta decicion, se limitaron las funcionalidades de copiar, cortar y pegar.

Nota 3: La funcionalidad de ordenar por columna (ColSort) saca provecho directamente de la desicion de projecto de que el valor de una celda siempre es representado como un “String”. De esta manera la comparacion entre celdas es equivalente a la comparacion de su valor los cuales son siempre compatibles.

3.5 Dato

Autor: Lluís Pujalte Feliu De Cabrera

Dato contiene una variable *valor* de tipo String que contiene el valor sin procesar de una celda.

Esta clase es la que se encarga de parsear (interpretar) los distintos valores de cada celda y devolver su contenido. Para el parseo se ha utilizado ANTLR4 el cual a partir de un archivo de gramática (Expr.g4) genera las distintas clases e interfaces necesarias para generar un Tree Visitor, *tokenizar* las expresiones, visitarlas y resolverlas en su debido orden. Concretamente las clases generadas son: ExprBaseLexer, ExprBaseVisitor, ExprBaseParser.

Funciones Clase Dato

Públicas:

- ***getValorProcesado*:** Retorna un double a partir de de *valor* que es resuelto por las clases anteriormente mencionadas por antlr.
- ***isRef*:** Método booleano que retorna true en el caso de que el *valor* del dato sea una referencia a otra celda.
- ***isNum:*** Función que retorna true en caso de que el parámetro sea un número .

Privadas:

- ***insertString:*** Método complementario usado para las fechas que inserta el *string stringToBeInserted* en la posición *index* del String *originalString.*
- ***validDate:*** Método booleano que retorna true en el caso de que el int *fecha* sea una fecha válida.
- ***incDate:*** Método que incrementa en uno la fecha pasada a la función.

3.5.1ExprBaseLexer

Analiza el string y lo divide en los distintos *Tokens* de la gramática. Un ejemplo de *Token* en la gramática que se ha creado sería el '+' o el NAME utilizado para identificar las funciones. Todos los tokens están almacenados en el archivo Expr.tokens.

3.5.2ExprParser

Genera el TreeVisitor De la expresión introducida a partir de los diferentes Tokens que el lexer ha obtenido previamente.

3.5.3ExprBaseVisitor

Visita el TreeVisitor generado por el parser y resuelve una por una las expresiones en el orden indicado. Esta clase es extendida por MyExprVisitor con la cual se define como

resolver cada una de las expresiones, cada una de estas tiene un método que sobreescribe al de la clase padre.

Funciones Clase MyExprVisitor

Públicas

- ***visitExponentExpr:*** Define como resolver una expresión del tipo exponencial.
  *Estructura de estas expresiones:*1.0^4.0

- ***visitFunctionDBExpr:*** Define como resolver una funcion con 4 parámetros que son dos Bloques de celdas.
  Estructura de estas expresiones (ejemplo): COV(~A:1,~C:6~,F:1,~F:6)

- ***visitNumericExpr:*** Define cómo resolver una expresión atómica (es decir solo un número) de tipo *double*.
  Estructura de estas expresiones (ejemplo): 2.0

- ***visitAddMinusExpr:*** Define cómo resolver una expresión con un + o un -.
  Estructura de estas expresiones (ejemplo): 3.0+~B:5

- ***visitMultDivExpr:*** Define cómo resolver una expresión con un / o un *.
  Estructura de estas expresiones (ejemplo): 5.0*AVG(~B:7,~D:5)

- ***visitParentExpr:*** Define como resolver las expresiones entre paréntesis.
  Estructura de estas expresiones (ejemplo): (5.0-6.0)

- ***visitFunctionExpr:*** Define cómo resolver una expresión de tipo funcional con un solo parámetro.
  Estructura de estas expresiones (ejemplo): SQRT(6.0)

- ***visitNumericNegExpr:*** Define cómo resolver una expresión de un número negativo.
  Estructura de estas expresiones (ejemplo): -1.0

- ***visitFunctionDExpr:*** Define cómo resolver una función con dos parámetros que representan un bloque de celdas.
  Estructura de estas expresiones (ejemplo): AVG(~B:7,~D:5)

- ***visitIdExpr:*** Define cómo resolver una expresión que sea una referencia a otra celda.
  Estructura de estas expresiones (ejemplo): ~B:6

Privadas

- ***getCol:*** Método complementario que devuelve la columna que representa el string que le pasas.

- ***resolve:*** Método utilizado para resolver las referencias a otras celdas.

Decisiones de diseño

Primeramente se decidió utilizar antlr4 debido a que de esta forma es mucho más simple añadir nuevas funcionalidades al proyecto y simplifica considerablemente problemas como el orden en el que resolver las expresiones (en antlr, se resuelven en el orden en el que las pongas en el archivo de gramática .g4) o el *parseo* de las mismas.

Seguidamente se optó por usar el carácter ":" para separar las coordenadas de las referencias a celdas ya que así con un simple *contains(":")* se podría determinar si la expresión contiene alguna referencia a otra celda (lo que ayudará al patrón observador de la clase Celda). Esto creó problemas con el *parseo* ya que no se dividían bien los tokens. La solución surgió de introducir otro token al inicio de la referencia, el carácter "~". Con esto finalmente las referencias a otras celdas vienen dadas por la estructura ~C:1.

Por otra parte las fechas las hemos tratado como números en los que los 4 primeros dígitos son el año, los 2 siguientes el mes y los 2 últimos el día. Estas al ser introducidas deben ir precedidas de una función de las dos posibles en este tipo de variables sea date o sea dateinc para así detectar que son de este tipo. Esto simplificó mucho la clase visitor ya que así todos los resultados del árbol serían doubles ya que en el caso de los strings no pasan por el tree visitor. Sin embargo a continuación se presentó el problema del tamaño máximo de los doubles, esto hizo que no se pudiera pasar al treeVisitor a no ser que cambiaramos el tipo de variable del árbol a otro más grande lo que a parte de que haría cambiar gran parte del código ralentizará mucho los tiempos de recorrido por este. Es por esto que finalmente la mejor solución que se presentó fue tratarlo fuera del arbolVisitor.

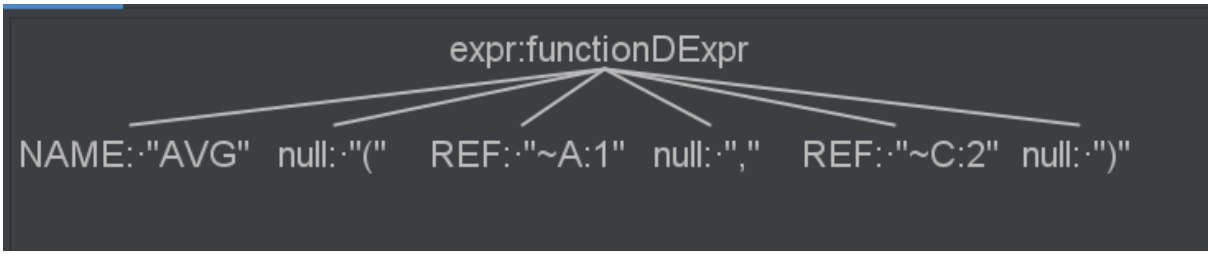
Los *strings* se comprueban antes de pasarlos a *antlr* y estos deben ir entre comillas dobles (ejemplo: "Hola").

Para las funciones se han decidido separar en 3 tipos dependiendo del número de operandos que tengan ya que así son más simples de tratar.

Todas las posibles expresiones y funciones que soporta la gramática Expr:

1.0	-5.6
1.0+2.1	3.0-4.0
2.0*5.0	5.0/2.3
4.0^2.3	3.3*(4.2-4.7)
~A:1	SQRT(3.0)
LN(2.0)	ROUND(5.6)
TRUNC(5.6)	INC(2.3)
DEC(1.4)	ABS(-1.0)
AVG(~A:1,~C:2)	MED(~A:1,~C:2)
VAR(~A:1,~C:2)	DESV(~A:1,~C:2)

Los parámetros de AVG, MED, VAR y DESV deben ser 2 referencias a celdas y estos realizan la operación indicada en este bloque de celdas, el árbol de *parsing* de uno de estos ejemplos sería:

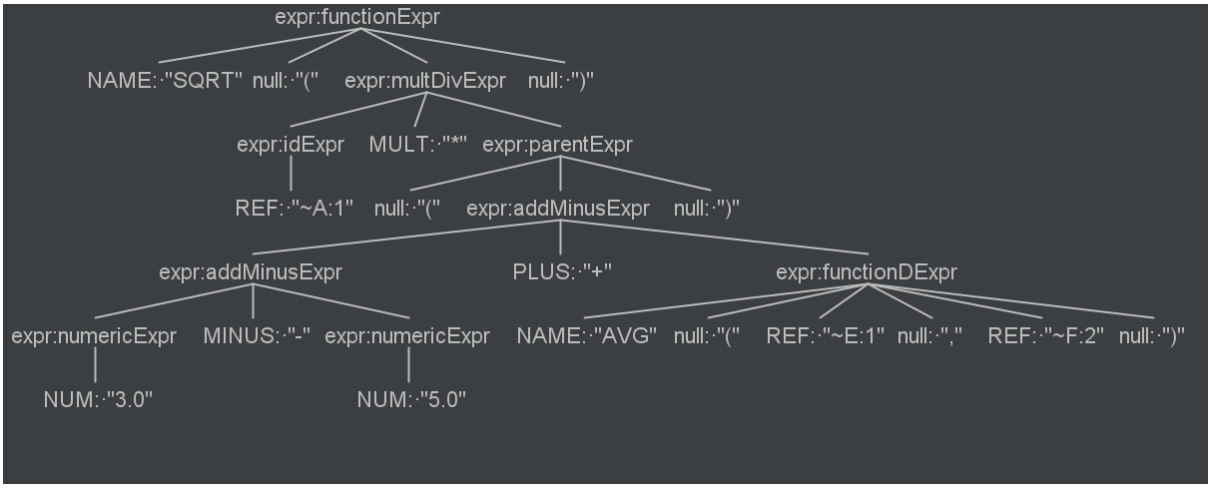


COV(~A:2,~D:6,~E:2,~H:6)
PEARS(~A:2,~D:6,~E:2,~H:6)

Los de COV y PEARS son dos bloques donde en el caso anterior las dos primeras referencias resultan el primero de estos bloques y las dos siguientes el segundo bloque. La tokenización de estas expresiones sería:

También soporta expresiones más complejas compuestas por las anteriores como por ejemplo: SQRT(~A:1*(3.0-5.0+AVG(~E:1,~F:2)))

El árbol de parseo generado por antlr de la expresión siguiendo este lenguaje sería el siguiente:



Para el control de errores como divisiones por 0, raíces negativas, etc. se ha decidido escribir en la celda en la que se ha producido el error: ERR. Esto una vez implementada la parte gráfica será fácilmente mejorable con mensajes de error específicos

para cada error sin embargo en esta entrega se ha decidido no usar estos mensajes ya que serían muy largos y harían difícilmente legible el *output* generado.

3.6 Estado

Autor: Alejandro Durán Saborido y Lluís Pujalte Feliu De Cabrera

La clase estado representa el estado actual del documento una vez abierto por el usuario. Contiene seis atributos estáticos, *HojaActual* apunta a la hoja que se encuentre abierta en todo momento, *BloqueActual* apunta al bloque sobre el cual se está trabajando (si lo hubiera) en todo momento, *CeldaCopiada* apunta a la celda que esté copiada (si lo hubiera), *BloqueCopiado* apunta al bloque que esté copiado (si lo hubiera) y por último *hayCeldaCopiada* y *hayBloqueCopiado* que son booleanos que indican en todo momento si hay alguna celda o bloque copiados respectivamente.

Se tomó la decisión de crear esta clase para agilizar la resolución de referencias a otras celdas en la clase Dato y para agilizar también la consulta del contenido de las celdas de la página sobre la cual se está trabajando, así como para implementar las funciones de copiar y pegar.

Funciones

Esta clase solo contiene *getters* y *setters* estáticos para todos sus atributos menos los dos booleanos que indican si hay una celda o un bloque copiado. Cabe destacar que el *setter* para *CeldaCopiada* y el *setter* para *BloqueCopiado* automáticamente vuelven cierto *hayCeldaCopiada* y *hayBloqueCopiado* al invocarse.