El proyecto de programación HULK, consiste básicamente, en crear un compilador para el lenguaje de programación HULK(Havana University Language for Kompilers). Para la realización de este, me basé en explotar la idea de la recursividad y de los árboles de sintaxis abstracta(AST), con la ayuda del paradigma de programación orientada a objetos.
Particularmente mi solución la idea que sigue es mediante una entrada del usuario, ir iterando por cada elemento de esta y separar cada uno como Tokens válidos, ir buscando hasta que encontrara una expresión válida para este lenguaje, y a la vez ir detectando errores los cuales se califican según su tipo como: Semánticos y de Sintaxis. Un Tokens es una secuencia de caracteres que representa una unidad semántica, como una palabra o un símbolo. Ejemplos de este pueden ser "+", "-", "let", "print", etc. Un AST, es una estructura de datos que representa la estructura sintáctica de un programa de computadora. El AST es generado por el compilador o intérprete de un lenguaje de programación y se utiliza para analizar y transformar el código fuente del programa. El AST es una representación abstracta del código fuente que facilita la manipulación y análisis del mismo. También me auxilio de otros conceptos importantes dentro de la compilación como Parser y Lexer. El Parser se encarga de analizar y procesar la estructura sintáctica de un lenguaje de programación. Su función principal es tomar el código fuente como entrada y determinar si cumple con las reglas de sintaxis del lenguaje, generando una representación estructurada del mismo. El Lexer es la primera fase en el proceso de compilación y se encarga de analizar el código fuente de un programa y dividirlo en tokens o unidades léxicas.
Para llevar a cabo esta solución cree una serie de clases las cuales tienen funciones específicas dentro de este proceso de compilación. A continuación hablaré detalladamente de cada clase en particular y sus funcionalidades dentro de este proyecto.
*Debido a problemas propios de mi pc tuve que cambiar varias reglas de sintaxis. A la hora de introducir números reales(doubles) estos no pueden contener puntos(ejemplo: 2.2), solo los reconoce de la siguiente manera(2,2 3,4). Debido a esto entonces el separador de argumentos usual(,) ha sido reemplazado por(:). Otro problema también es a la hora de los strings que en vez de usar las doble comillas(" algo "), fue reemplazado por las simples(' algo '). De no seguir estas pautas a la hora de ejecución saltará un error, disculpen las molestias ocasionadas.

Clase Token

La clase pública Token está compuesta por campos de clase o variables, los cuales su función es almacenar los datos de los Tokens, ya sea su tipo y valor, para así poder ser usados posteriormente en distintas etapas del proyecto.

Campos de Clase:

- string Type;
- object Value;

El constructor de esta clase recibe como parámetros un string para rellenar el Type y un object para rellenar el Value y así construir un objeto de tipo Token, con todos sus datos dentro de él. Esto es muy importante ya que así puedo revisar los errores luego y además voy dándole un sentido a la entrada del usuario para poder comprender y estructurar lo que pide el mismo.

Clase Let

La clase estática Let está compuesta por campos de clases y funciones, las cuales me ayudan a Parsear el Token de tipo let. Este se usa para declarar variables que pueden ser de tipo string o numéricas, como int o double y realizar operaciones aritméticas con ellas, en el caso de las numéricas, dentro del cuerpo del let, ya que fuera de este, las variables dejan de existir.

Campos de Clase:

- List(string) variables;
- List(string) valores

Con estos campos lo que hago es almacenar los identificadores de las variables en la primera lista y en la otra los valores correspondientes a las mismas. Para esto voy iterando por todo el let, buscando siempre el último let por si está anidado, y separándolo en dos partes. La primera es donde se encuentra la declaración de las variables, antes del in, dejándolo solo con los identificadores y los valores para así almacenarlos correctamente. En esta etapa también miro a ver si encuentro algún error y si lo encuentro lo agrego a una lista de errores(hablaré más adelante de esto). Las variables el usuario las debe entrar separadas por(:). La siguiente etapa es verificar el cuerpo del let, después

del in, donde se encuentran las operaciones a realizar. Aquí sustituyo, donde aparezca la variable, por su valor, para devolver el string solo con números, modificando la entrada inicial, para más facilidad a la hora de evaluar la expresión dentro del let. Igual compruebo que no exista errores de ningún tipo para poder parsear la expresión más primitiva. Para todo esto creé una serie de métodos que me permiten realizar en análisis anterior

Funciones:

- bool let(string text);
- string letin(string text);

El primero devuelve true si dentro de la expresión pasada, el let se encuentra correctamente declarado(solo compruebo si la palabra let está escrita como se debe) auxiliándome de buscar la posición del let, que si está mal declarado va a devolver -1 y false el método. En el segundo es donde analizo toda la declaración del let. Primero busco el índice del let y del in, y creo un nuevo string desde el final de la palabra let hasta el principio del in, separándolo por los(:), si existen, para sacar cada variable con su valor. Luego pico del in hasta el final para analizar el cuerpo y cambiar las variables por su valor. Este método devuelve la expresión dentro del cuerpo del let sin variables, solo con sus valores y símbolos aritméticos, para ser evaluada posteriormente. Obviamente aquí también busco errores de todos los tipos y si existen voy rellenando la lista de errores, si no continua la ejecución normal

Clase Print

La clase estática Print es la encargada de analizar los Tokens de tipo Print. Esta comprueba que realmente el Token pasado sea de tipo Print y realiza el trabajo necesario para devolver una expresión, en caso de existir, o un texto en caso de estar el interior del cuerpo del print entre comillas, para ser mostrado en consola al usuario. Para esto tengo unas funciones encargadas de realizar todas estas operaciones.

Funciones:

- bool printChek(string text);
- string print (string text);
- bool Parentesis(string expresion);

En el primero compruebo si realmente el Token print no contiene ningún error, devolviendo verdadero en este caso y si es así continuo para el segundo método, de lo contrario devuelve falso. El segundo primeramente comprueba si lo que está dentro del cuerpo del print es una expresión o un texto. En caso de ser este último devuelve este intacto, del otro modo devuelve la expresión para ser evaluada posteriormente. Esta diferencia es posible ya que primero busco si posee comillas y pico por ahí, de lo contrario es una expresión y la mantengo. Este método, igual que en let, cambia la entrada del usuario por lo que encontró y lo devuelve, para más facilidad a la hora de evaluar la expresión o simplemente imprimir el texto entre comillas. Igual aquí compruebo que no tenga errores de ningún tipo y si se encuentran relleno la lista de errores para al final devolverlos. El método Parentesis(), es un método auxiliar que uso para comprobar, que si la entrada contiene parentesis, los mismos estén balanceados. Este lo llamo en distintos métodos y momentos donde hace falta revisar esto.

Clase Error

La clase pública Error es la encarga de almacenar y construir los distintos objetos de tipo Error, los cuales tienen un tipo, una posición y un argumento(el texto a mostrar explicándolo), que se almacenan en los distintos campos de clase.

Campos de Clase:

- int ErrorPos;
- string Argument;
- enum ErrorType
- ErrorType errorType;

El constructor de esta clase recibe como parámetros el tipo de error correspondiente almacenados dentro del enum, la posición donde se encuentra y el argumento explicando el error. Así crea objetos de tipo Error para ser almacenados, si se encuentran, en la lista de errores.

Clase Parser:

La clase estática Parser es la encargada de, a partir de la entrada del usuario, separar esta en Tokens dependiendo de su tipo y realizar las funciones cor-

respondientes. También verifica que los Tokens sean válidos y no contengan errores de ningún tipo. En esta clase se analizan expresiones del tipo algebraicas $(+, -, /, *, Pow)$, y booleanas $(<, >, |, and)$ y si existe alguna de otro tipo, pues se llama a la clase correspondiente donde es analizada. Luego de todo este proceso, se chequea aquí que la lista de errores no contenga ningún elemento, de ser así, devuelve el resultado de evaluar la expresión marcada por el usuario.

Campos de Clase:

- string text;
- int pos;
- Token currentToken;
- char currentChar;
- List(Error) ErrorList;

La entrada del usuario es pasada primeramente por el método (string Write(string test)), este devuelve el resultado final de evaluar la expresión. Pero primeramente toma dicha entrada y la va separando por Tokens, llamando al método(Token GetToken()). Este decide que tipo de Token representa el caracter que se encuentra en la entrada en una posición x, si encuentra Tokens que no son del tipo algebraicos o booleanos pues llama a las funciones encargadas de su análisis. Luego de evaluarlos, comprueba que no existan errores y dependiendo de los Tokens validos es lo que devuelve el método.

Clase PI:

La clase PI estática, es la encargada de, que si se encontró un Token de tipo PI, analizarlo y comprobar que no contenga errores y devolver el valor de dicho número PI.

Funciones:

- bool PICheck(string text);
- string pi(string text);

Con el primero compruebo que PI, no contenga ningún error sintáctico, de ser así llamo al segundo que, donde se encuentre PI en el texto, lo sustituye por su valor numérico y devuelve el texto modificado para seguir siendo analizado.

Clase Sin:

La clase estática Sin es la encargada de analizar los Tokens de tipo "Sin", el seno(x). Esta comprueba primeramente que dicho Token no contenga ningún error y de ser así realiza las operaciones necesarias para, donde se encontraba este Token en el texto, sustituirlo por su valor numérico y devolver dicho texto modificado.

Funciones:

- bool sinCheck(string text);
- string sin(string text);

Con el método (string sin(string text)), comprueba que no contenga errores de ningún tipo y toma el argumento dentro de los paréntesis y lo evalúa, ya que puede contener otra expresión dentro del argumento. Al final calcula el seno de la expresión y sustituye en el texto, donde aparezca este Token, por su valor numérico, y es lo que devuelve este método, el texto modificado.

Clase Cos:

La clase estática Cos es la encargada de analizar los Tokens de tipo "Cos", el coseno(x). Esta comprueba primeramente que dicho Token no contenga ningún error y de ser así realiza las operaciones necesarias para, donde se encontraba este Token en el texto, sustituirlo por su valor numérico y devolver dicho texto modificado.

Funciones:

- bool cosCheck(string text);
- string cos(string text);

Con el método (string cos(string text)), comprueba que no contenga errores de ningún tipo y toma el argumento dentro de los paréntesis y lo evalúa, ya que puede contener otra expresión dentro del argumento. Al final calcula el coseno de la expresión y sustituye en el texto, donde aparezca este Token, por su valor numérico, y es lo que devuelve este método, el texto modificado.

Clase Log:

La clase estática Log es la encargada de evaluar y analizar, si existen, las funciones de tipo logarítmicas. Para ello analiza primero que el Token no contenga ningún error, de ser así, revisa cada argumento del logaritmo para verificar que tipo de expresión es y evaluarla, en caso de ser válida.

Funciones:

- bool logCheck(string text);
- string log(string text);

Esto es posible gracias al método(string log(string text)), el cual separa los argumentos del logaritmo y analiza los mismos dependiendo del tipo de expresión que contenga y si es válida dentro de este. Al final, si no se encontró ningún error, devuelve la entrada inicial modificada, donde se encontraba el logaritmo lo sustituye por su valor numérico final. Cabe destacar que un error dentro de este método, es que los argumentos contengan números negativos o que el resultado de evaluar alguna expresión dentro de estos devuelva un valor negativo, ya que esto es una regla fundamental de las funciones logarítmicas.