INFORME PROYECTO HULK

HULK es un lenguaje de programación imperativo, funcional, estática y fuertemente tipado. Casi todas las instrucciones en HULK son expresiones. En particular, el subconjunto de HULK que usted implementar se compone solamente de expresiones que pueden escribirse en una línea.

Este Proyecto esta compuesto por varias clases cada cual con su función:

Clase BinaryMyExpression:

La clase tiene dos propiedades, LeftMyExpression y RightMyExpression, que representan los valores de las expresiones izquierda y derecha respectivamente. Estas propiedades son de tipo MyExpression.

La clase también tiene una propiedad abstracta "value" que devuelve el valor de la expresión. Esta propiedad está marcada como abstracta, lo que significa que cualquier clase que herede de BinaryMyExpression debe proporcionar una implementación para esta propiedad.

La clase tiene un constructor que toma dos parámetros de tipo MyExpression y los asigna a las propiedades LeftMyExpression y RightMyExpression.

La clase también tiene un método ToString() que devuelve una representación en forma de cadena de la expresión binaria.

Clase Context

La cual tiene las siguientes características

- Un campo público llamado myVars que es una lista de objetos de tipo Variable.
- Un constructor sin parámetros que inicializa la lista myVars y agrega una variable llamada "PI" con el valor de Pi en la lista.
- Un método público llamado AddVar que se utiliza para agregar una variable a la lista myVars. Si la variable ya existe en la lista, se lanza una excepción de tipo SyntaxException.
- Un método público llamado FindVar que se utiliza para buscar una variable en la lista myVars utilizando su nombre. El método devuelve un objeto de tipo Variable si se encuentra la variable y devuelve null si no se encuentra.

Clase KeyWord:

La clase tiene los siguientes elementos:

- Una lista de palabras clave públicas llamada `Keywords`, que contiene las palabras "let", "in" y "function".
- Una variable privada llamada `_value` de tipo `string`, que almacena el valor de una palabra clave.
- Una propiedad pública llamada `value`, que permite acceder al valor de la palabra clave almacenada en `_value`.
- Un constructor público llamado `KeyWord`, que recibe un valor de tipo `string` y asigna ese valor a `_value`.

Clase MyExpression

El siguiente código define una clase abstracta llamada MyExpression. Esta clase tiene tres métodos abstractos:

- El método Evaluate, que devuelve una cadena de caracteres y no tiene implementación.
- La propiedad value, que devuelve una cadena de caracteres y no tiene implementación.
- El método ToString, que devuelve una cadena de caracteres y no tiene implementación.

Esta clase abstracta se utiliza como base para implementar expresiones personalizadas en una aplicación. Al heredar de esta clase, se deben implementar los métodos abstractos según sea necesario para cada expresión.

Clase Node:

El código define una interfaz llamada "Node". Esta interfaz declara una propiedad de solo lectura llamada "value" que debe ser de tipo "string". La interfaz "Node" puede ser implementada por cualquier clase y hace que la clase implementadora tenga una propiedad llamada "value" que devuelve un valor de tipo "string".

PredFunction:

El código define una clase abstracta llamada "PredFunction" que hereda de otra clase llamada "MyExpression".

La clase "PredFunction" tiene los siguientes miembros:

- "Args": una lista de argumentos.
- "CantArgs": una propiedad abstracta que devuelve la cantidad de argumentos que debe tener la función.
- "value": una propiedad abstracta que devuelve el valor de la función.
- Un constructor que recibe un valor y una lista de argumentos. Si la cantidad de argumentos recibidos no coincide con la cantidad de argumentos esperada por la función, se lanza una excepción de sintaxis.
- "Evaluate": un método abstracto que debe ser implementado en las clases derivadas y que devuelve una cadena de texto.
- "value": una propiedad que devuelve el valor de la función.

Esta clase define una plantilla básica para crear otras clases que representen predicados de funciones específicas.

Token:

El código define una clase llamada "Token" que representa un token en un lenguaje de programación. Un token es una unidad léxica individual que se utiliza en el análisis léxico de un programa.

La clase Token tiene dos propiedades: "value", que almacena el valor del token como una cadena de texto, y "type", que almacena el tipo del token como un valor de la enumeración "TokenType".

La enumeración "TokenType" enumera diferentes tipos de tokens que pueden aparecer en un programa, como números, símbolos matemáticos, palabras clave, identificadores, operadores lógicos, etc.

En resumen, este código define una clase paraar tokens en un lenguaje de programación y enumera los posibles tipos de tokens que pueden existir.

UnaryExpression:

El siguiente código define una clase abstracta llamada `UnaryExpression` que hereda de la clase `MyExpression`.

La clase `UnaryExpression` tiene un constructor protegido que recibe un parámetro `value` y asigna ese valor a la propiedad `this.value`.

La clase también tiene un método abstracto `Evaluate()` que se espera que las clases hijas implementen. Este método devuelve un valor de tipo `string`.

Además, la clase `UnaryExpression` tiene una propiedad `value` que obtiene el valor asignado en el constructor y un método `ToString()` que devuelve el valor de la propiedad `value` en forma de cadena.

UserFunction:

Este código define una clase llamada `UserFunction` que representa una función definida por el usuario. Esta clase tiene las siguientes propiedades y métodos:

- `Name`: variable de tipo `string` que almacena el nombre de la función.
- `FunBody`: variable de tipo `List` que almacena el cuerpo de la función (una lista de instrucciones o elementos).
- `ParamNames`: variable de tipo `List` que almacena los nombres de los parámetros de la función.
- `UserFunctions`: variable estática de tipo `List` que almacena todas las funciones definidas por el usuario.

El método estático `AddFunction` recibe una instancia de `UserFunction` como argumento y verifica si ya existe una función con el mismo nombre en la lista `UserFunctions`. Si no existe, agrega la función a la lista.

El método estático `FindFunction` recibe el nombre de una función como argumento y busca en la lista `UserFunctions` una función con ese nombre. Si encuentra una función con ese nombre, la devuelve, de lo contrario, devuelve `null`.

Variable:

El siguiente código define una clase llamada "Variable", que tiene dos propiedades: "Name" y "VarTree". La propiedad "Name" es de tipo "string" y

representa el nombre de la variable. La propiedad "VarTree" es de tipo "MyExpression" y representa la expresión asociada a la variable.

Además, la clase tiene un constructor que recibe dos parámetros: "name" y "varTree". El constructor asigna los valores de los parámetros a las propiedades "Name" y "VarTree" de la clase.

Y la gran JOYA de la corona 😊

Clase Parser:

El analizador sintáctico se define como una clase llamada "Parser" que tiene las siguientes propiedades y métodos:

- "current" es una variable entera que mantiene un seguimiento del índice actual en la lista de tokens.
- "tokens" es una lista que contiene los tokens obtenidos del analizador léxico.
- "tree" es una variable de tipo "MyExpression" que almacena la estructura de datos generada por el analizador sintáctico.
- "context" es una variable de tipo "Context" que almacena la información del contexto en el que se está ejecutando el programa.

El método "Evaluate" devuelve una cadena de texto que representa la evaluación de la expresión generada por el analizador sintáctico.

El método "Parse" es el núcleo del analizador sintáctico. Verifica el tipo del token actual y realiza diferentes acciones en base a ese tipo. Si el token es del tipo "VarDeclarationKeyWord", se llama al método "ParseVarDeclaration" para analizar una declaración de variable. Si el token es del tipo "If", se llama al método "ParseIfDeclaration" para analizar una declaración condicional. En caso contrario, se llama al método "ParseAndOr" para analizar una expresión lógica o de comparación.

El método "ParseFunDeclaration" se utiliza para analizar una declaración de función. Primero se verifica el tipo del token actual y se extrae el nombre de la función. Luego se obtienen los parámetros de la función llamando al método "GetFunDeclParams". Por último, se verifica que haya un token del tipo "Arrow" y se obtiene el cuerpo de la función. La función resultante se agrega a una lista de funciones definidas por el usuario.

El método "GetFunDeclParams" se utiliza para obtener los parámetros de una declaración de función. Se recorren los tokens a partir del siguiente token al de apertura de paréntesis y se van agregando los nombres de las variables a una lista. Si se encuentra una coma, se continúa con la siguiente variable. Cuando se encuentra el token de cierre de paréntesis, se devuelve la lista de nombres de variables.

El método "ParseIfDeclaration" se utiliza para analizar una declaración condicional if-else. Primero se verifica que el token actual sea de tipo "OpenParenthesis". Luego se resuelven los paréntesis llamando al método "SolveParenthesis" y se evalúa su resultado. Si el resultado es "true", se analiza el bloque de código del "if" hasta encontrar el token "Else". Si el

resultado es "false", se salta el bloque de código del "if" y se busca el token "Else". Una vez encontrado el token "Else", se crea una nueva instancia de la clase "Parser" para analizar el bloque de código del "else" y se obtiene su árbol de expresión.

El método "SolveParenthesis" se utiliza para resolver una expresión dentro de paréntesis. Se recorren los tokens a partir del siguiente token al de apertura de paréntesis y se van agregando a una lista. Al encontrar el token de cierre de paréntesis, se crea una nueva instancia de la clase "Parser" para analizar la lista de tokens y se obtiene su árbol de expresión.