Informe del Hulk

Kendry Javier del Pino Barbosa

October 13, 2023

1 Introducción

Este informe describe el funcionamiento un de un compilador realizado en el lenguaje C# para el lenguaje `Hulk`. A continuación se mostrará el funcionamiento de la biblioteca de clases formada para su correcto funcionamiento.

2 Clase Lexer

La clase Lexer se encarga de analizar una cadena de código fuente y dividirla en tokens. A continuación, se describen los principales componentes y métodos de la clase:

- `private string input`: Un campo privado que almacena la cadena de código fuente de entrada.
- `public List<Token> tokens`: Una lista que almacena los tokens encontrados durante el análisis.
- `private int currentPosition`: Un campo que mantiene un seguimiento de la posición actual en la cadena de entrada durante el análisis.
- `public Lexer(string input)`: El constructor de la clase que recibe la cadena de código fuente como parámetro e inicializa los campos de la clase.
- `public Token[] Tokenize()`: El método principal que inicia el proceso de análisis léxico. Recorre la cadena de entrada carácter por carácter y genera tokens según las reglas definidas.
- `private bool IsUnaryOperator()`: Un método privado que verifica si el carácter actual es un operador unario válido.

- `private Token ScanNumber()`: Un método privado que escanea y reconoce un número en la cadena de entrada.
- `private Token ScanIdentifierOrKeyword()`: Un método privado que escanea y reconoce identificadores o palabras clave en la cadena de entrada.
- `private bool IsOperator(char c)`: Un método privado que verifica si un carácter es un operador válido.
- `private bool MatchArrow()`: Un método privado que verifica si el carácter actual es parte de la flecha "=¿".
- `private bool MatchKeyword(string keyword)`: Un método privado que verifica si la cadena actual coincide con una palabra clave específica.
- `private Token ScanUnaryOperator()`: Un método privado que escanea y reconoce operadores unarios en la cadena de entrada.
- `private Token ScanString()`: Un método privado que escanea y reconoce cadenas de texto en la cadena de entrada.

3 Clase Token

La clase Token contiene una enumeración de tipos de tokens y una clase Token para representar los tokens. A continuación, se describen los elementos de la clase:

- `enum TokenType`: Una enumeración que define los tipos de tokens utilizados en el analizador léxico. Estos tipos incluyen números, identificadores, operadores, palabras clave y otros.
- `class Token`: Una clase que representa un token y tiene propiedades para el tipo de token, su valor y la longitud del valor.
- `public Token(TokenType type, string value)`: El constructor de la clase Token que recibe el tipo y el valor del token y los asigna a las propiedades de la clase.
- `public override string ToString()`: Un método sobrecargado que devuelve una representación en cadena del token en el formato (Tipo, 'Valor').

4 Clase Parser

La clase Parser se encuentra en el namespace Hulk y tiene la responsabilidad de analizar tokens y generar el árbol sintáctico de un programa. A continuación, se describen los elementos de la clase:

- `private Lexer lexer`: Un campo privado que almacena una instancia de la clase Lexer para analizar tokens.
- `private Token[] tokens`: Un campo privado que almacena la lista de tokens generada por el analizador léxico.
- `private int currentTokenIndex`: Un campo privado que lleva un seguimiento del índice del token actual durante el análisis.
- `private Dictionary<string, Node> variables`: Un diccionario que almacena variables y sus valores.
- `private Dictionary<string, FunctionDeclarationNode> userDefinedFunctions`: Un diccionario que almacena funciones definidas por el usuario.
- `List<string> parameters`: Una lista que almacena los parámetros de la función actualmente en análisis.
- `public Parser(Lexer lexer, Dictionary<string, FunctionDeclarationNode> userDefinedFunctions)`: El constructor de la clase que recibe una instancia de Lexer y un diccionario de funciones definidas por el usuario.

- `public Node Parse()`: Un método público que inicia el análisis y devuelve un nodo raíz del árbol sintáctico.
- `private Node ParseStatements()`: Un método privado que analiza y devuelve una secuencia de declaraciones.
- `private Node ParseStatement()`: Un método privado que analiza y devuelve una declaración.
- `private Node ParseFunctionDeclaration()`: Un método privado que analiza y devuelve una declaración de función definida por el usuario.
- `private Node ParseExpression()`: Un método privado que analiza y devuelve una expresión.
- `private Node ParseLetIn()`: Un método privado que analiza y devuelve una declaración LetIn.
- `private Node ParseTerm()`: Un método privado que analiza y devuelve un término en una expresión.
- `private Node ParseParenthesizedExpression()`: Un método privado que analiza y devuelve una expresión entre paréntesis.
- `private Node ParseFactor()`: Un método privado que analiza y devuelve un factor en una expresión.
- `private bool Match(TokenType type, string value = null)`: Un método privado que verifica si el token actual coincide con un tipo y un valor opcional.
- `private bool Expect(TokenType type, string value = null)`: Un método privado que verifica si el token actual coincide con un tipo y un valor opcional, y lanza una excepción si no coincide.
- `private Token Consume()`: Un método privado que consume el token actual y avanza al siguiente.

5 Clase Node

La clase abstracta Node es la clase base en la jerarquía de clases relacionadas con el análisis y evaluación de nodos en un árbol sintáctico. Define dos métodos clave:

- `public abstract object Evaluate()`: Este método es abstracto y debe ser implementado por las clases derivadas. Representa la lógica para evaluar un nodo particular en el árbol sintáctico y devuelve un resultado.
- `public virtual object EvaluateWithVariables(Dictionary<string, Node> variables)`: Este método virtual proporciona una forma de evaluar un nodo teniendo en cuenta un diccionario de variables. Por defecto, simplemente llama a `Evaluate()`, pero las clases derivadas pueden sobrescribirlo para considerar variables.

6 Clase - NumberNode : Node

La clase **- NumberNode** hereda de la clase Node y se utiliza para representar nodos que contienen números. Tiene un campo Value que almacena un token numérico.

- `public Token Value ( get; )`: Un campo que almacena el valor numérico en forma de token.
- `public NumberNode(Token value)`: Un constructor que recibe un token numérico y lo asigna al campo `Value`.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase Node. En este caso, realiza la evaluación del nodo numérico y devuelve el valor como un entero si es convertible, como un número decimal si es convertible o como una cadena si no es convertible a número.

7 Clase - StringNode : Node

La clase **- StringNode** hereda de la clase Node y se utiliza para representar nodos que contienen cadenas de texto. Tiene un campo Value que almacena un token de cadena.

- `public Token Value (get;)`: Un campo que almacena el valor de la cadena en forma de token.
- `public StringNode(Token value)`: Un constructor que recibe un token de cadena y lo asigna al campo `Value`.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase Node. En este caso, simplemente devuelve el valor de la cadena tal como está, sin cambios.
- `public override object EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase Node. Al igual que el método `Evaluate()`, devuelve el valor de la cadena tal como está, sin cambios. Esta implementación considera un diccionario de variables, pero no las utiliza en la evaluación de la cadena.

8 Clase - VariableNode : Node

La clase **- VariableNode** hereda de la clase Node y se utiliza para representar nodos que contienen nombres de variables. Tiene un campo VariableName que almacena un token de nombre de variable.

- `public Token VariableName (get;)`: Un campo que almacena el nombre de la variable en forma de token.
- `public VariableNode(Token variableName)`: Un constructor que recibe un token de nombre de variable y lo asigna al campo `VariableName`.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase Node. En este caso, simplemente devuelve el token de nombre de variable.
- `public override object EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase Node. En este método, se busca el valor de la variable en el diccionario de variables proporcionado como argumento. Si la variable está definida, devuelve el valor correspondiente. Si no está definida, se lanza una excepción indicando que la variable no está definida.

9 Clase - UnaryOperationNode : Node

La clase **- UnaryOperationNode** hereda de la clase Node y se utiliza para representar nodos que contienen operaciones unarias. Tiene un campo Operator que almacena el operador unario y un campo Right que representa el operando de la operación.

- `public string Operator (get;)`: Un campo que almacena el operador unario.
- `public Node Right (get;)`: Un campo que almacena el operando de la operación.
- `public UnaryOperationNode(string oper, Node right)`: Un constructor que recibe el operador unario y el operando de la operación y los asigna a los campos `Operator` y `Right` respectivamente.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase Node. En este método, se evalúa el operando derecho y se aplica el operador unario correspondiente. En este caso, el operador puede ser el signo negativo ("-"), y se verifica si el operando es un número entero o decimal para aplicar la operación adecuada. El resultado se devuelve como un objeto.

10 Clase - MathFunctionNode : Node

La clase **- MathFunctionNode** hereda de la clase Node y se utiliza para representar nodos que contienen funciones matemáticas. Tiene un campo FunctionName que almacena el nombre de la función y un campo Argument que representa el argumento de la función.

- `public string FunctionName (get;)`: Un campo que almacena el nombre de la función.
- `public Node Argument (get;)`: Un campo que almacena el argumento de la función.
- `public MathFunctionNode(string functionName, Node argument)`: Un constructor que recibe el nombre de la función y el argumento de la función, y los asigna a los campos `FunctionName` y `Argument` respectivamente.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase Node. En este método, se evalúa el argumento de la función y se aplican las funciones matemáticas correspondientes. Las funciones admitidas son "cos" (coseno), "sen" (seno) y "log" (logaritmo natural). Se verifica el nombre de la función y se calcula el resultado utilizando la biblioteca Math de C#. El resultado se devuelve como un objeto.
- `public override object EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase Node. En este método, se evalúa el argumento de la función con variables y se aplican las funciones matemáticas correspondientes de manera similar al método `Evaluate()`. El resultado se devuelve como un objeto.

11 Clase - PrintNode : - Node

La clase **- PrintNode** hereda de la clase **- Node** y se utiliza para representar nodos que imprimen valores en la consola. Tiene un campo Expression que representa la expresión que se imprimirá.

- `public Node Expression (get;)`: Un campo que almacena la expresión que se imprimirá.
- `public PrintNode(Node expression)`: Un constructor que recibe la expresión y la asigna al campo `Expression`.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase **- Node**. En este método, se evalúa la expresión y se convierte el resultado en una cadena (string) que se imprime en la consola. Luego, el método devuelve el resultado como una cadena.
- `public override object EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase **- Node**. En este método, se evalúa la expresión con variables si las hubiera. El resultado se mantiene sin cambios y se devuelve como está.

La clase **- LogFunctionNode** hereda de la clase **- Node** y se utiliza para representar nodos que calculan el logaritmo de un número en una base dada. Tiene dos campos, BaseExpression que representa la base del logaritmo y NumberExpression que representa el número al que se aplica el logaritmo.

- `public Node BaseExpression (get;)`: Un campo que almacena la expresión que representa la base del logaritmo.
- `public Node NumberExpression (get;)`: Un campo que almacena la expresión que representa el número al que se aplica el logaritmo.
- `public LogFunctionNode(Node baseExpression, Node numberExpression)`: Un constructor que recibe la base del logaritmo y el número al que se aplica el logaritmo, y los asigna a los campos `BaseExpression` y `NumberExpression` respectivamente.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase **- Node**. En este método, se evalúan las expresiones de la base y el número, y se calcula el logaritmo en la base dada del número. El resultado se devuelve como un objeto.
- `public override object EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase **- Node**. En este método, se evalúan las expresiones de la base y el número con variables si las hubiera. Luego, se calcula el logaritmo en la base dada del número. El resultado se devuelve como un objeto.

12 Clase - LetInNode : - Node

La clase **- LetInNode** hereda de la clase **- Node** y se utiliza para representar nodos que declaran variables locales y luego ejecutan una declaración. Tiene dos campos, DeclaredVariables que almacena las variables declaradas como un diccionario y Statement que representa la declaración que se ejecutará.

- `public Dictionary<string, Node> DeclaredVariables (get;)`: Un campo que almacena las variables locales declaradas como un diccionario. La clave es el nombre de la variable y el valor es la expresión asociada.
- `public Node Statement (get;)`: Un campo que almacena la declaración que se ejecutará después de declarar las variables locales.
- `public LetInNode(Dictionary<string, Node> declaredVariables, Node statement)`: Un constructor que recibe el diccionario de variables declaradas y la declaración a ejecutar, y los asigna a los campos `DeclaredVariables` y `Statement` respectivamente.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase **- Node**. En este método, se crea un nuevo diccionario para las variables evaluadas, se evalúan cada expresión de variable y se agregan al nuevo diccionario. Luego, se ejecuta la declaración `Statement`. El resultado de la declaración se devuelve como un objeto.

13 Clase - IfElseNode : - Node

La clase **- IfElseNode** hereda de la clase **- Node** y se utiliza para representar nodos condicionales que ejecutan una rama si una condición es verdadera y otra rama si la condición es falsa. Tiene tres campos: Condition que representa la condición a evaluar, IfBranch que es la rama que se ejecutará si la condición es verdadera, y ElseBranch que es la rama que se ejecutará si la condición es falsa.

- `public Node Condition (get;)` : Un campo que almacena la condición a evaluar.
- `public Node IfBranch (get;)`: Un campo que almacena la rama que se ejecutará si la condición es verdadera.
- `public Node ElseBranch (get;)`: Un campo que almacena la rama que se ejecutará si la condición es falsa.
- `public IfElseNode(Node condition, Node ifBranch, Node elseBranch)`: Un constructor que recibe la condición, la rama para el caso verdadero y la rama para el caso falso, y las asigna a los campos `Condition`, `IfBranch` y `ElseBranch` respectivamente.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase **- Node**. En este método, se evalúa la condición y se verifica si el resultado es un valor booleano. Si la condición es verdadera, se ejecuta la rama `IfBranch`, y si es falsa, se ejecuta la rama `ElseBranch`. El resultado de la rama ejecutada se devuelve como un objeto. Si la condición no es un valor booleano, se lanza una excepción.
- `public override object EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase **- Node**. En este método, se evalúa la condición con variables y se aplica la lógica de ejecución condicional de manera similar al método `Evaluate()`. El resultado de la rama ejecutada se devuelve como un objeto. Si la condición no es un valor booleano, se lanza una excepción.

14 Clase - FunctionDeclarationNode : - Node

La clase **- FunctionDeclarationNode** hereda de la clase **- Node** y se utiliza para representar la declaración de una función en un lenguaje de programación. Tiene tres campos: Name que almacena el nombre de la función, Parameters que es una lista de parámetros de la función y Body que representa el cuerpo de la función.

- `public string Name ( get;)`: Un campo que almacena el nombre de la función.

- `public List<string> Parameters (get;)`: Un campo que almacena la lista de parámetros de la función.
- `public Node Body (get; )`: Un campo que almacena el cuerpo de la función.
- `public FunctionDeclarationNode(string name, List<string> parameters, Node body)`: Un constructor que recibe el nombre de la función, la lista de parámetros y el cuerpo de la función, y los asigna a los campos `Name`, `Parameters` y `Body` respectivamente.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase **- Node**. En este método, no se evalúa la función en este momento; se devuelve la propia instancia de la declaración de la función. La evaluación real de la función se realiza en su llamada.
- `public override object EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase **- Node**. En este método, se evalúa el cuerpo de la función en el contexto de las variables proporcionadas. Se crea un nuevo diccionario para las variables locales de la función y se agregan los parámetros al diccionario local. Luego, se evalúa el cuerpo de la función en el contexto local y se devuelve el resultado.

15 Clase - BooleanNode : - Node

La clase **- BooleanNode** hereda de la clase **- Node** y se utiliza para representar nodos que contienen expresiones booleanas en un lenguaje de programación. Tiene tres campos: left que representa la expresión izquierda, operatorToken que almacena el operador de comparación y right que representa la expresión derecha.

- `public BooleanNode(Node left, Token operatorToken, Node right)`: Un constructor que recibe la expresión izquierda, el token del operador y la expresión derecha, y los asigna a los campos `left`, `operatorToken` y `right` respectivamente.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase **- Node**. En este método, se evalúan las expresiones en el momento necesario. Se convierten las expresiones izquierda y derecha a valores numéricos y se realiza la comparación según el operador. Los operadores admitidos son "¿", "¡", "==", "!=", "¿=", "¡=". El resultado de la comparación se devuelve como un valor booleano.
- `public override object EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase **- Node**. En este método, se evalúan las expresiones izquierda y derecha en el contexto de las variables proporcionadas. Luego, se realiza la comparación de manera similar a `Evaluate()`. El resultado de la comparación se devuelve como un valor booleano.

16 Clase - BinaryOperationNode : - Node

La clase **- BinaryOperationNode** hereda de la clase **- Node** y se utiliza para representar nodos que contienen operaciones binarias en un lenguaje de programación. Tiene tres campos: Left que representa la expresión izquierda, Operator que almacena el operador de la operación y Right que representa la expresión derecha.

- `public BinaryOperationNode(Node left, Token operatorToken, Node right)`: Un constructor que recibe la expresión izquierda, el token del operador y la expresión derecha, y los asigna a los campos `Left`, `Operator` y `Right` respectivamente.
- `public override object Evaluate()`: Implementa el método `Evaluate()` heredado de la clase **- Node**. En este método, se evalúan las expresiones izquierda y derecha en el momento necesario. Se realiza la operación binaria según el operador. Los operadores admitidos son "+", "-", "
- `public override Node EvaluateWithVariables(Dictionary<string, Node> variables)`: Implementa el método `EvaluateWithVariables()` heredado de la clase **- Node**. En lugar de evaluar los valores de las expresiones en este método, se crea un nuevo objeto **- BinaryOperationNode** con los nodos izquierdo y derecho actualizados con los resultados de la evaluación en el contexto de las variables proporcionadas.

17 Clase - FunctionCallNode: - Node

- `FunctionName` (string): Almacena el nombre de la función a llamar.
- `Arguments` (List¡Node¿): Contiene los argumentos que se pasan a la función.
- `userDefinedFunctions` (Dictionary¡string, FunctionDeclarationNode¿): Un diccionario que mapea nombres de funciones a sus definiciones.
- `functionCallStack` (Stack¡Dictionary¡object, Node¿¿): Una pila de diccionarios de variables locales.
- `Evaluate`: Este método evalúa la llamada a la función.
- `EvaluateWithVariables`: Este método permite evaluar la función con un conjunto personalizado de variables. Este método se utiliza principalmente para llamadas recursivas y funciona de manera similar al método `Evaluate`.

18 Clase Main

La clase Main es la clase principal de una aplicación en C# que utiliza el analizador léxico y el analizador sintáctico para procesar y ejecutar código. Esta clase contiene el punto de entrada de la aplicación y coordina el proceso de análisis y ejecución del código fuente. A continuación, se describen los principales componentes y métodos de la clase:

- `static void Main(string[] args)`: Este es el método principal de la clase y es el punto de entrada de la aplicación. A continuación, se describe su funcionamiento:
  - :Este método comienza mostrando un diseño estético en la consola mediante la llamada a `DisplayHulkCompilerBan` El banner de la aplicación se muestra en morado sobre un fondo negro.
  - Variables locales: En este método se declara una variable `userDefinedFunctions` de tipo `Dictionary<string, Fu` que se utiliza para almacenar las funciones definidas por el usuario. Este diccionario asocia nombres de funciones con objetos de tipo `FunctionDeclarationNode`.
  - Bucle infinito: Luego, se inicia un bucle infinito con `while (true)` que permite que la aplicación acepte entrada del usuario de forma continua hasta que el usuario ingrese "exit". El bucle se encarga de realizar la siguiente operación:
    - Interfaz de usuario: Establece el color del texto en morado y muestra un indicador "¿¿ " en la consola para indicar que la aplicación está esperando una entrada del usuario.
    - Entrada de usuario: Lee la entrada del usuario desde la consola mediante `Console.ReadLine()` y almacena la entrada en la variable `input`. Si el usuario ingresa "exit", el bucle se interrumpe y la aplicación sale.
    - Análisis léxico y sintáctico: Se intenta realizar el análisis léxico y sintáctico del código fuente ingresado por el usuario. Se crea un objeto `Lexer` y un objeto `Parser` para llevar a cabo estos procesos. Se parsea la entrada del usuario y se obtiene un árbol de sintaxis abstracta (`Node`).
    - Evaluación: Se evalúa el árbol de sintaxis abstracta obtenido y se almacena el resultado en la variable `result`.
    - Salida: Se muestra el resultado en la consola en color amarillo. Dependiendo del tipo de resultado (`int`, `double`, `string`, etc.), se muestra de manera apropiada.
    - Manejo de funciones definidas por el usuario: Si el resultado es una declaración de función, se agrega al diccionario `userDefinedFunctions` asociando el nombre de la función con su definición.
    - Manejo de errores: En caso de que se produzca una excepción durante el análisis o la evaluación, se captura la excepción y se muestra un mensaje de error en rojo en la consola.
- `static void DisplayHulkCompilerBanner()`: Este método se encarga de mostrar un banner de bienvenida estilizado en la consola. El banner se muestra en morado sobre un fondo negro y contiene el nombre de la aplicación.