Havana University Language for Kompilers (HULK) Interpreter

Segundo Proyecto de Programación Primer Año de Ciencias de la Computación

Vladimir Piñera Verdecia C-112+1

14 de octubre de 2023

Resumen

Este informe presenta el desarrollo de un intérprete del lenguaje de programación HULK. El intérprete fue creado en una aplicación de consola de .NET 7.

1. Introducción

En este informe, se describe el proceso de diseño e implementación de la aplicación de consola **HULK Interpreter** y se detallan las características principales de un intérprete.

El informe también incluye ejemplos de uso de la aplicación de consola **HULK Interpreter** y se detallan los pasos necesarios para ejecutar la aplicación en diferentes sistemas operativos.

2. El lenguaje HULK

HULK es un lenguaje de programación imperativo, funcional, estático y fuertemente tipado. La mayoría de las construcciones sintácticas en HULK son expresiones, incluyendo el cuerpo de todas las funciones, bucles y cualquier otro bloque de código. El cuerpo de un programa en HULK siempre termina con una única expresión global (y con un punto y coma final) que sirve como punto de entrada del programa. Esto significa que, por supuesto, un programa en HULK puede consistir en una sola expresión global.

2.1. Subconjunto de HULK que fue implementado

La documentación completa del lenguaje puede encontrarla en este link
HULK Interpreter soporta instrucciones que pueden ser ejecutadas en una sola línea y se implementó el siguiente subconjunto del lenguaje:

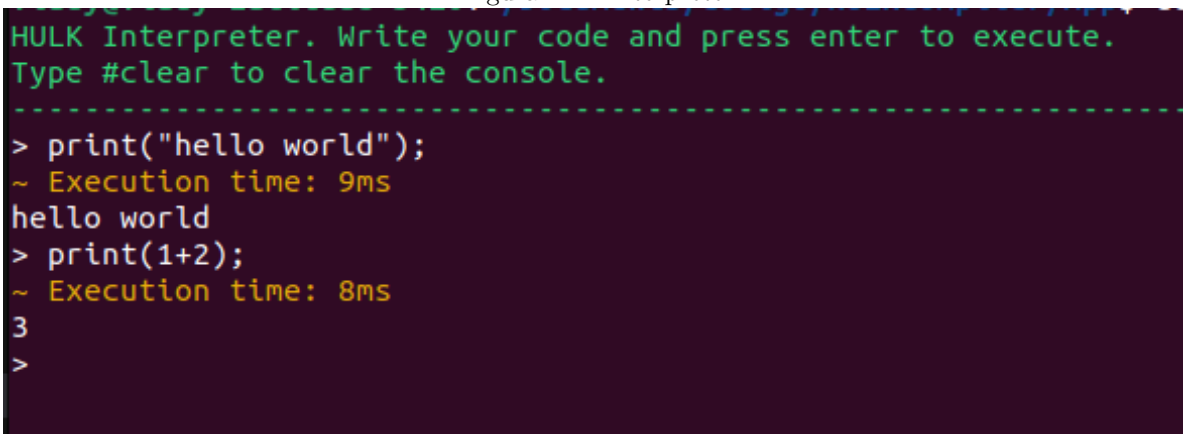
- Tipos de datos:
  - string
  - number
  - boolean
  - *vector (sintaxis explícita y soporte para los métodos next, current y size, así como el indexado con los corchetes)
- Expresiones aritméticas
- Función *print()*
- Saltos de línea, tabulación, concatenación de strings e inclusión de comillas dentro de la declaración de un string
- Funciones matemáticas y constantes predefinidas
  - *sqrt()
  - sin()
  - cos()
  - *exp()
  - log(base, valor)
  - *rand()
  - PI
  - *E
- Definición de funciones
  - Soporte para funciones recursivas
- Variables
- Condicionales
  - *Soporte para *elif*
- *Bucles
  - for
  - while
- Inferencia de tipos

- Con asterisco (*) las funcionalidades extra

3. La aplicación HULK Interpreter

El intérprete es una aplicación de consola en la cual puedes ejecutar código HULK, luego de escribir una expresión válida y presionar la tecla Enter podrá ver el resultado (así como el tiempo de ejecución) y en caso de haber algún error aparecerá en consola. En caso de querer limpiar la consola, escriba *#clear*.

Figura 1: El intérprete



4. Estructura del Proyecto

El proyecto está divido en dos partes fundamentales: La app de consola y la biblioteca de clases (carpeta "lib") donde se encuentra toda la lógica del intérprete. Esa carpeta tiene la siguiente estructura

- Errors
  - Errors.cs
- Expressions
  - AtomicExpressions.cs
  - ForExpression.cs
  - FunctionCallExpression.cs
  - IfExpression.cs
  - LetInExpression.cs
  - Node.cs
  - OtherExpressions.cs
  - UnaryBinaryExpressions.cs
  - WhileExpression.cs
- Lexer
  - Lexer.cs
  - Token.cs
- Parser
  - Parser.cs
  - SyntaxTree.cs
- Semantic
  - Evaluator.cs
  - InferenceTypes.cs

A continuación se detalla el contenido de algunas de estas clases

4.1. Errors

Una pequeña clase para almacenar los errores en cada período del intérprete, tiene métodos para comprobar si hay algún error, para mostrarlo, para añadirlo, o para removerlo.

4.2. Expressions

Aquí se encuentran todas las expresiones que soporta el intérprete, cada clase de expresión hereda de una clase Expression y esta a su vez de una clase llamada Node. Cada expresión específica tiene una propiedad para saber su tipo, además de un método para evaluar esa expresión.

4.3. Lexer

En la clase Lexer se analiza la entrada del usuario y se devuelve una lista de tokens, cada token válido para el intérprete se encuentra almacenado en el archivo Token.cs y la clase Token tiene propiedades útiles para cada token como son su tipo, valor o posición dentro del código.

4.4. Parser

Aquí se encuentra la clase Parser (la más extensa del proyecto) que tiene métodos para matchear y analizar cada token con el fin de devolver la expresión correspondiente.

4.5. Semantic

La clase Evaluator es la encargada de evaluar cada expresión y tiene propiedades estáticas útiles como Diagnostics para almacenar los errores, ScopePointer para tener referencia del scope en el que son creadas las variables, FunctionBody para comprobar la correcta declaración del cuerpo de una función, así como dos propiedades para almacenar el scope de variables y funciones. La clase InferenceTypes infiere el tipo de dato de una expresión, esto es útil, por ejemplo, para que no se llame a una función con un tipo de dato incorrecto.

5. Flujo de trabajo del intérprete

El intérprete pasa por 3 fases fundamentales:

- Análisis Léxico
- Análisis Sintáctico
- Análisis Semántico

Cuando se escribe una instrucción en el intérprete, se hace un llamado al método estático Parse(string text) de la clase SyntaxTree (encargada de almacenar el árbol sintáctico que devuelve el analizador sintáctico). El método Parse() crea una instancia de la clase Parser (encargada del análisis sintáctico) y luego llama al método Parse() de esta clase. Dentro de este se crea una instancia de clase Lexer para identificar los tokens, luego de esto usando la técnica del parsing recursivo, el parser analiza sintácticamente los tokens y devuelve las expresiones correspondientes. Al finalizar este proceso, si ocurrió algún error, se mostraría en la consola con un mensaje especificando el tipo de error, una descripción y la posición donde ocurrió, en caso contrario se pasaría al análisis semántico llamando al método Evaluate(node) de la clase Evaluator. Dentro de ese método dependiendo del tipo de expresión que sea se comprueba que no hayan errores semánticos (como variables o funciones sin declarar) y se evalúa para así devolver el resultado. Si lo ingresado por el usuario fue una declaración de función sólo se comprobaría que no haya ningún error y se guardaría la función dentro del scope para ser utilizada luego, o sea, que no se mostraría nada en pantalla.

6. Manejo de Errores

El intérprete maneja 3 tipos de errores fundamentales:

- Errores Léxicos
- Errores Sintácticos
- Errores Semánticos

También maneja algunos errores en tiempo de ejecución (como *Stack Overflow*).

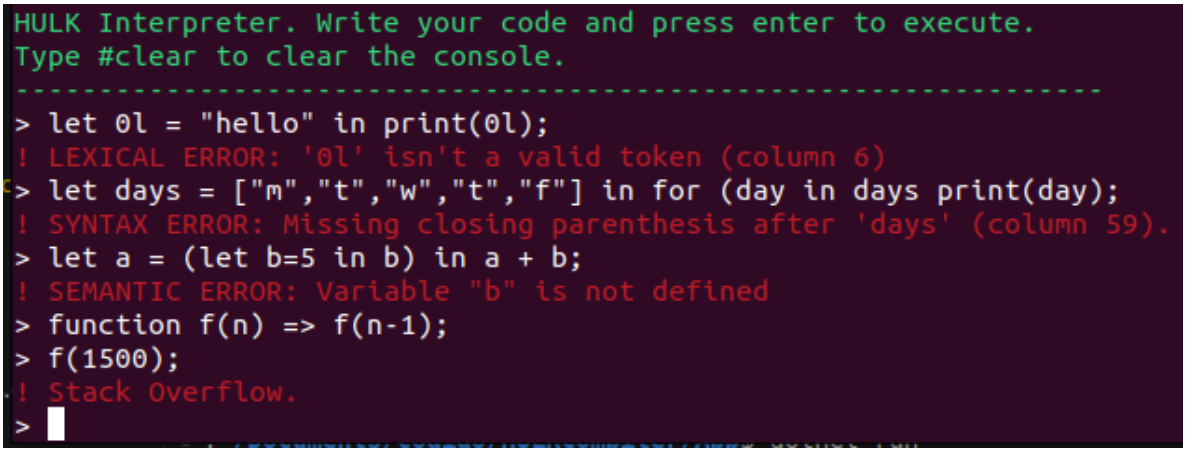


Figura 2: Errores

Figura 3: Ejemplos de código