

HULK (Proyecto de Programación II)

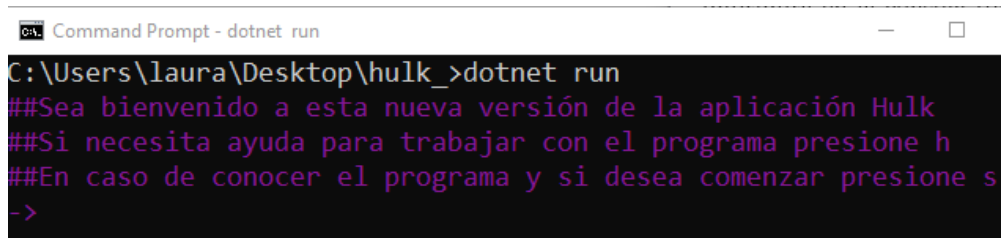
Laura Alonso Rivero C-113

¿Qué es HULK ?

‘**HULK**’, o Havana University Language by Kompilers, por sus siglas en inglés es un lenguaje de programación imperativo, funcional, estática y fuertemente tipado. En este proyecto se ha implementado un subconjunto de **HULK** el cual se compone solamente de expresiones que pueden escribirse en una línea. El intérprete de **HULK** será una aplicación de consola, donde el usuario puede introducir una expresión presionar **ENTER**, e inmediatamente se verá el resultado de evaluar la expresión en caso de que lo hubiera. El proceso se continuará ocurriendo hasta que el usuario presione **Ctrl + C**.

Al iniciar la aplicación el usuario tiene dos opciones:

- En caso de no conocer el funcionamiento de la aplicación puede presionar ‘ h ’ y se imprimirá en la consola todas las posibles expresiones que puede declarar con este lenguaje.
- Si conoce el funcionamiento y si desea comenzar puede presionar ‘ s ’ de este modo se inicia el proceso de compilación.



```
Command Prompt - dotnet run
C:\Users\laura\Desktop\hulk_>dotnet run
##Sea bienvenido a esta nueva versión de la aplicación Hulk
##Si necesita ayuda para trabajar con el programa presione h
##En caso de conocer el programa y si desea comenzar presione s
->
```

Figure 1: Opciones

El proyecto consta de tres pasos fundamentales en su proceso de compilación:

- Lectura de la expresión introducida por el usuario. De este proceso se encarga el **Lexer** el cual escanea la entrada y devuelve una lista de **Tokens**.
- Creación del Árbol de Sintaxis Abstracta, AST por sus siglas en inglés. Esta parte corre por el **Parser** que devuelve una lista donde se clasifican las diferentes expresiones.
- La interpretación de la expresión. Llevada a cabo por el **Interpreter** que va a devolver la evaluación de la expresión entrada por el usuario.

Lectura de la expresión (Ánàlisis Léxico)

Cuando el usuario introduce la línea comienza la lectura de la expresión, con este propósito se creó la clase **Lexer**. Esta clase consta de un método principal llamado **Scan** el cual recorre cada caracter de la entrada del usuario y dependiendo de sus valores se irán clasificando. Esta clasificación está dada debido al tipo de **Token** que sea.

Un **Token** es una secuencia de caracteres que posee un valor y tipo específico. Existen diversos tipos como son: las palabras claves (print, let, if), los nombres de las variables o las funciones (x, y, fib), los operadores aritméticos, de relación y lógicos (*, ==, &), las funciones matemáticas (cos, sqrt, log) y finalmente los literales numéricos, cadenas de strings o booleanos. Estos están agrupados en un enum nombrado **TypesOfToken**. Para guardar todos los tokens específicos del lenguaje se creó un diccionario **Tokens** que posee como valor el tipo de **Token** que es y su valor y cuya llave es un string que será el valor del **Token**.

Una vez escaneada toda la entrada se devolverá una lista con todos estos tokens en la posición en que fueron encontrados para su posterior procesamiento.

Creación del AST (Án lisis Sint ctico)

A partir de la lista de Tokens retornada se procede a crear el Árbol Sintáctico el cual no es más que una forma de definir los pasos a seguir por el Compilador a la hora de evaluar las expresiones de una forma recursiva. En este caso se uso el método de parsing recursivo descendente, el cual consiste en construir el árbol a partir de las reglas de gramática de cada lenguaje. Esta gramática se encuentra implementada en la clase **Parser** y sigue la siguiente estructura:

```

SuperiorExpression = function id (GetArguments) => ToDeclare | ToDeclare ;
ToDeclare = let Assignment in ToDeclare | Conditional;
Conditional = if (ToDeclare) ToDeclare else ToDeclare | Logical;
Logical = Equal ( ( '|' | '&' ) Equal );
Equal = Compare ( ( '==' | '!=' ) Compare );
Compare = Term ( ( '<' | '<=' | '>' | '>=' ) Term );
Term = Factor ( ( '+' | '-' | '@' ) Factor );
Factor = Power ( ( '/' | '*' | '%' ) Power );
Power = Unary ( ( '^' ) Power );
Unary = ( '!' | '-' | '+' | '(' | "print" ) | MathFuction;
MathFunction = ( "log" | "sin" | "cos" | "sqrt" | "expo" | "rand" ) | Literal;
Literal = Números | cadenas de strings | true | false | PI | E | id;

```

Figure 2: Gramática

Con este método de parsing se comienza a partir del símbolo inicial (raíz) y se va descendiendo en el AST hacia las hojas. En cada caso se elige la regla gramatical adecuada y se invoca la función correspondiente para continuar el análisis.

Para lograr la generación del árbol fue necesario crear la clase abstracta **Expression** para poder identificar los distintos tipos de expresiones que el usuario era capaz de implementar y agregarlas a una lista. Al ser abstracta toda aquella clase que la herede podrá contar con los métodos que posee. En este caso el único método a heredar es utilizado en la evaluación de las expresiones.

Interpretación de la expresión (Análisis Semántico)

En esta etapa se interpreta, evalúa y finalmente se devuelve el resultado de la entrada del usuario a través de la clase **Interpreter**. El objetivo es verificar que las operaciones sean realizadas con los operandos correctos.

En algunas expresiones como la declaración de variables y funciones fue necesaria la creación de un **Scope global**. Esto no es más que el conjunto de variables o funciones que son accesibles desde cualquier parte del programa. Es muy importante para evitar conflictos y ambigüedades en el código pues una vez declarada una función o variable no es posible declarar una con el mismo nombre en el mismo ámbito y con las mismas características

Para interpretar las expresiones se invoca el método **Evaluate** el cual como su nombre indica realiza la evaluación de los distintos tipos de expresiones guardados en la lista. Para ello se implementó la interfaz **Helper** que genera distintos métodos para evaluar cada expresión según sus particularidades.

Una vez evaluada la entrada el resultado se imprimirá en la consola.

Errores

En el lenguaje HULK existen tres tipos distintos de errores que se producen en cada etapa del proceso:

Error Léxico: se presenta cuando las expresiones contienen caracteres no válidos en el lenguaje.

Error Sintáctico: se presenta cuando la secuencia de tokens no es válida atendiendo a la gramática del lenguaje o cuando se trata de sobrescribir una variable o función en un mismo contexto.

Error Semántico: se presenta cuando no se puede evaluar la expresión debido a incoherencias entre la operación y los tipos de expresiones que se encuentra en ella.

Con el correcto funcionamiento de esta aplicación considerada un compilador de una versión más simple del lenguaje **HULK**, se puede afirmar la capacidad para leer, interpretar y evaluar operaciones con instrucciones que posean cualquier tipo de las estructuras permitidas (operación aritmética, llamado de funciones, declaración de funciones, condicionales, e incluso evaluar funciones con implementación recursiva).