HULK

(Havana University Language for Kompilers)

Victor Hugo Pacheco Fonseca

8 de octubre del 2023

1. Introducción

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

2. Que puede hacer

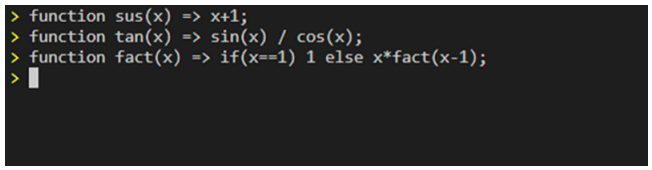
2.1. Expresiones básicas

Todas las expresiones en Hulk deben terminar en ;. Cuenta con las constantes matemáticas PI y E , además de los operadores matemáticos básicos:

Además cuenta con funciones matemáticas tales como: sin(x), cos(x), log(x,y) ,sqrt(x), pow(x,y) y rand() ,donde sin y cos reciben su parámetro en radiales.
Los string en Hulk también esta definidos y deben ir entre doble comillas. Estos puede ser concatenados con otros string (o la representación en string de números) a traves del operador @.

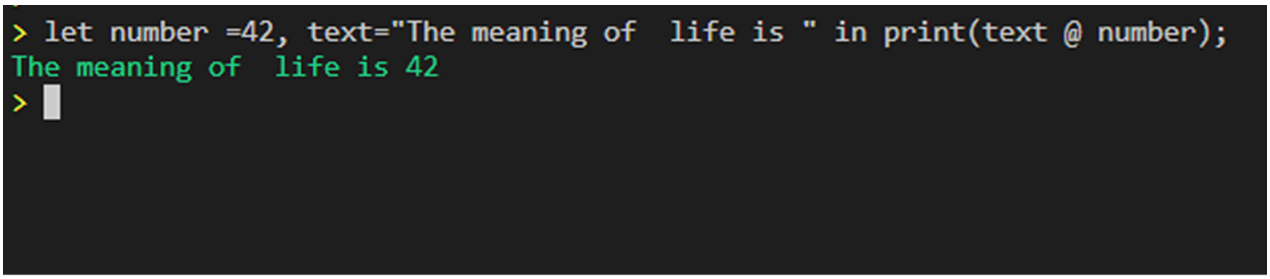
2.2. Funciones

En Hulk se pueden declarar funciones inline. Las cuales puede ser definidas de la siguiente forma: *function* name(operadores) ⇒ *cuerpo de la funcion*; Estas funciones pueden ser simples,compuestas y recursivas.



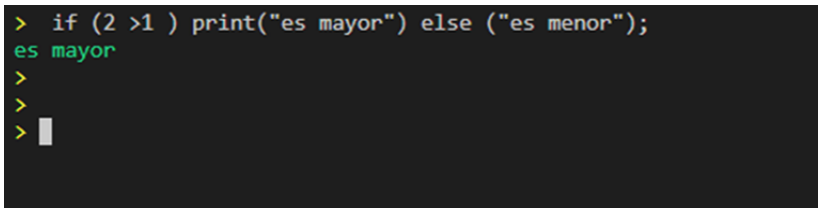
2.3. Variables

En Hulk es posible declarar variables usando la expresion let-in, mediante la sintaxis: *let* identifier = value *in* expression ;
En general, una expresion let-in consta de una o mas delaraciones de variables, y fuera de la expresión let-in dejan de existir.



2.4. Condicionales

Las condiciones en Hulk se implemetan mediante una expresión if-else, que debe recibir una expresión boleana entre parentesis, y dos expresiones para el cuerpo de if y del else(siempre ambas partes).



3. Funcionamiento

3.1. Lexer

La primera fase del compilador es el analizador léxico.Durante esta etapa el código fuente se divide en unidades léxicas llamadas tokens, estos representan componentes básicos del lenguaje : palabras reservadas, identificadores, números, strings y operadores. El encargado de esto es la clase Lexer, con su método Scanner. Su objetivo es identificar los diferentes tokens, crear un objeto de tipo Token donde se almacena su valor, posición y tipo, para luego almacenarlos en una lista.

3.1.1. Tipos de tokens

Los tokens se clasifican en :

1. NumberToken
2. StringToken
3. PlusToken
4. MinusToken
5. MultiplicationToken
6. DivisionToken
7. RestoToken
8. ExponentToken
9. ArrobaToken
10. AndOperatorToken
11. OrOperatorToken
12. NotOperatorToken
13. EqualEqualToken

14. NotEqualToken
15. ComparisonToken
16. AssignmentToken
17. OpenParenthesesToken
18. CloseParenthesesToken
19. ReservedWordToken
20. ReservedFunctionToken
21. IdentifierToken
22. ColonToken
23. SemiColonToken
24. EndLineToken

3.2. Parser

La segunda fase del compilador es el análisis sintáctico, también conocido como parsing. Este recibe la lista de tokens del Lexer y verifica que se cumplan las reglas sintácticas establecidas y se construye una estructura de datos (árbol sintáctico) que representa la sintáxis del programa. De esto se encarga la clase Parser, en la cual se encuentran diversos métodos que retornan un objeto tipo Expression (clase de la cual heredan muchas otras) donde cada uno revisa un operador diferente, y de forma tal que los métodos que verfican operadores de mayor precedencia se llamen dentro de los de menor precedencia, asegurandose asi que en la creación del árbol un nodo (operador) de menor precedencia siempre esta arriba de uno de mayor precedencia. Por ejemplo para el codigo 2*3+1 :
Pirmero se llama al metodo ParseTerm() el cual verifca los operadores '+' y '-', pero antes llama al metodo ParseProduc() que verifica si hay alguno de los operadores '*' y '/' , y retorna un objeto tipo BinaryExpression que hereda de la clase Expression , que no es mas que una estructura de datos donde tiene un nodo el cual es un Token operador, y dos campos de clases de tipo Expression, left y right, en este caso retornaria BinaryExpression(2 , *

, 3).Hecho esto la funcion ParseTerm() identifica el siguente token '+' y retorna un objeto tipo BinaryExpression donde se almacena el arbol sintactico correspondiente.

3.3. *Evaluator*

Terminado el análisis sintáctico, la tercera fase es el análisis semántico, el cual verificar que no halla errores semánticos como por ejemplo una suma de expresiones que no sean tipo números .De esto se encarga la clase Execute, el método Evaluator(). Dicho método evalua recursivamante el árbol sintáctico mientras va desciendiendo en él.