

# Hulki

A Hulk interpreter.

*User Manual*

```
> let be = "Welcome" , to="Hulk" in be @ " to " @ to;  
Welcome to Hulk  
> █
```

## What is this document about?

This document is provided with the HulkI interpreter and it teaches the basics of the Hulk language and aspects of the interpreter users should know to have a better experience.

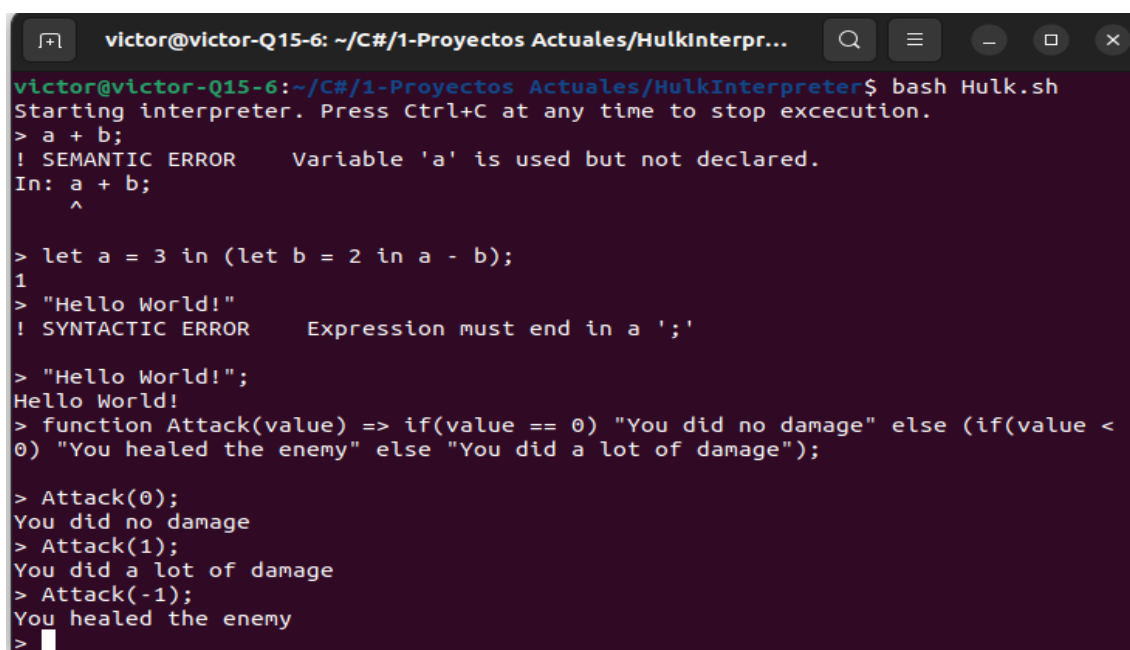
## What is HulkI?

HulkI is a Hulk interpreter, that is, a program created for executing Hulk source code.

Hulk is a programming language created at Habana University for educational purposes. You can learn more about it at <https://matcom.in/hulk>.

## How to use HulkI?

HulkI is designed to be used interactively on a console. You type some code and it executes it instantly, awesome, isn't it.



```
victor@victor-Q15-6: ~/C#/1-Proyectos Actuales/HulkInterpr...
victor@victor-Q15-6:~/C#/1-Proyectos Actuales/HulkInterpreter$ bash Hulk.sh
Starting interpreter. Press Ctrl+C at any time to stop execution.
> a + b;
! SEMANTIC ERROR      Variable 'a' is used but not declared.
In: a + b;
    ^

> let a = 3 in (let b = 2 in a - b);
1
> "Hello World!";
! SYNTACTIC ERROR      Expression must end in a ';'

> "Hello World!";
Hello World!
> function Attack(value) => if(value == 0) "You did no damage" else (if(value <
0) "You healed the enemy" else "You did a lot of damage");

> Attack(0);
You did no damage
> Attack(1);
You did a lot of damage
> Attack(-1);
You healed the enemy
>
```

Figure 1: HulkI interpreter running on a console on Ubuntu

To use Hulk on a Linux environment just call the Hulk.sh script.

This script executes the interpreter by default. If you call it with the parameter clean it erases the folders created during compilation, notice that if you execute it again they will be recreated. If you call it with the parameter help it shows a help message.

There is no .bat script yet but its being worked on. Meanwhile Windows users can use this command:

**dotnet run --project HulkInterpreter.csproj**

It should be called from the same folder as the HulkInterpreter.csproj file.

Its required that you have installed the .NET SDK to build the program from its source files. You can download it at <https://dotnet.microsoft.com/en-us/download/dotnet/7.0> .

This program was developed and tested using .NET SDK version 7.0.102

### Learning Hulk by examples:

At this point the interpreter should be flashing on your screen. Now lets use it.

First , every expression on Hulk ends in a semicolon ‘;’

Second, all expressions consist of a single line. Support for multiple line source code will be added in the future.

On Hulk you can use numbers :

2;

-1;

14.57;

They are all treated as real numbers.

You can do the usual operations with them:

$2 * 5;$      //Multiplication  
 $3 / 1.5;$     //Division  
 $4 + 1;$      //Sum  
 $6 - 2;$      //Subtraction  
 $2 ^ 3;$      //Exponentiation  
 $4 \% 2;$     //Reminder of the division

The precedence of the operations is as follows:

- 1-) ^
- 2-) \* / %
- 3-) + -

Lower number operations are executed before higher number operations. Operations of the same precedence are executed from left to right.

The associativity of the operations is as follows:

Left	Right
+ - * % /	^

Left associativity means operations are done from left to right.

Right associativity means operations are done from right to left.

As an example:

$2 - 3 - 4$  is executed as  $(2-3)-4$

$2 ^ 3 ^ 4$  is executed as  $2 ^ (3 ^ 4)$

Those operations can only be used with numbers.

You can also put a – in front of a number to indicate its opposite.

$-3;$   
 $-27.1;$

```
--43;    //Which is 43, not 42.
```

There are also two special constants.

```
PI;      //Which is 3.14....
```

```
E;      //Which is 2.71....
```

On Hulk you can use strings:

```
"pedrito";
```

```
"pedrito clavo un clavito";
```

```
"Hello World!";
```

```
"27";
```

```
"This is on one line \n and this is on another line";
```

```
"I am \t tabbed";
```

```
"\"I am quoted\"";
```

The last three examples shows the usage of escaped characters.

\n for new lines.

\t for tabs.

\ " for quotes.

Strings can be concatenated with the @ operator.

```
"alpha" @ "bet";    //This yields alphabet
```

That means the string to the right is put right after the string on the left.

On Hulk you can use booleans:

Which can have two possible values: true , false

true;

false;

Besides you can use the following operators, whose result are boolean values.

On numbers only:

`2 > 3;     //Yields false`

`2 < 3;     //Yields true`

`2 >= -E;   //Yields true`

`2 <= 27.1 //Yields true`

These are called relations : > , >= , < , <=

They are not associative. You can only use one operator:

`2 >= 3;    //Ok`

`2 <= 3 <= 4; //Error`

On booleans only:

`! true;     //Yields false`

`! false;    //Yields true`

`!(2 > 3);   //Yields true because 2 > 3 is false.`

! is the negation operator which return the opposite truth value.

! has right associativity, the rightmost is executed first.

On any type:

```
2 == 3;    //Yields true
```

```
2 == "Samantha";    //Yields false
```

```
"Peter" != true;    //Yields true because they are different
```

These are called equality operators. == , !=

== for equality

!= for not equality

They are not associative. You can only use one operator.

```
2 == 3;    //Ok
```

```
2 == 3 == 2;    //Error
```

You can also use logical operators:

& for logical and

| for logical or

```
2 + 3 < 4 & true;
```

```
"flies" == "bugs" | 2 < 3;
```

```
true & true;
```

```
false | true;
```

They have left associativity.

On Hulk you can also use parentheses to alter the order on which operations are executed.

```
2 * (2 + 1);
```

$2 ^ (3 / 1.5);$

*Table 1: Precedence*

Order	Operations	Comment
1	()	
2	! -	Unary
3	^	
4	* / %	
5	> >= < <=	
6	!= ==	
7	&	Logical and
8		Logical or

## Variables:

On Hukl you can also use variables.

```
let x = 5 in x * 2;
```

```
let x = "drink" , y = "wine" in (let z = "more" in x @ z @ y );
```

As shown on the examples above, variables are declared using a let-in expression.

```
let <variable declarations> in <some expression>;
```

Note that you must declare at least one variable.

```
let in 2;    //This is an error
```

```
let x = 2 in 2;    //This is ok
```

Variable declarations are straightforward:

```
<name> = <expression>
```

Where name is the variable name and expression is the value of the variable or an expression, which evaluates to a value.

```
let d = 2 * PI, r = 5 in d * r;    //Assign the result of an expression to
```



//a variable.

In case you do something like this:

```
let a = 2 in (let a = 3 in a * 10); //The result is 30
```

The innermost 'a' is what define the value of the variable, this is called scoping.

As you may have noticed, if several variables are declared you must separate them by a comma.

```
let var1 = 1, var2 = 2, var3 = 4 in var3 ^ var2 ^ var1;
```

## What is an expression?

At this point you may be asking yourself what is an expression.

An expression on Hulk is anything that evaluates to a number, a string or a boolean, simplifying, anything that has a value.

Every code snippet you have seen so far is an expression. Can you tell its type?

For arithmetic operators the value of the expression is the value of the operation.

For string concatenation is the concatenation of the strings.

For operators like < <= > >= == != their value is either true or false.

For a let-in expression its value is the value of the expression after the in.

That allows us to do this:

```
let a = 5 in a * (let a = 2 in a * a); //This is 20
```

The value of the leftmost let-in is 5 times the value of the innermost let-in, which is 4.

## Conditional Expressions

To make decisions on code we use an if-else expression.

```
let x = 2 in if(x < 2) "To low" else "Fine";
```

The structure of an if-else expression is:

```
if (<condition>) <expression1> else <expression2>
```

Condition must be a boolean expression, that is, it must be either true or false.

If its true then the value of the if-else is the value of expression1, if not then the value of the if-else is the value of expression2.

Condition, expression1 and expression2 are necessary, omitting any is an error. The parentheses are also necessary.

```
If (2 > 3) else "Is false"; //Error
```

At this point you have the basics to begin using Hukl.

But there is one more element to learn.

## Functions

Functions are useful as we can use them to write code once and then reuse it.

Consider the following function.

```
function max(a,b) => if(a > b) a else b;
```

After its declared we can know the maximum element very easily.

```
let x = 5, y = -12 in max(x,y);
```

The first example is the declaration of the function, the second example is the call to the function.

Function declarations have the following structure:

```
function <name> (<arguments>) => <expression>;
```

Where name is the name of the function. Notice that after you declare a function you cant declare variables with the same name.

```
function sum(a,b) => a + b;
```

```
let sum = 2 in sum;      //Error, you cant declare a variable with  
                        //a function name
```

Arguments is a list of comma separated variable names. Those can be used on the function body, which is the expression after the => symbol.

```
function f() => "I have no arguments";
```

```
function f(a) => "I have one argument";
```

```
function f(a,b) => "I have two arguments";
```

```
function f() => "I am similar to first function f";  //This is an error
```

On HulkI you cant redefine a function with the same name and arguments as a previously declared function.

Some functions come by default on the language.

Those are:

```
rand()    //Generate a random number between 0 and 1
```

```
cos(x)    //Calculates the cosine of x, x is a number
```

```
exp(x)    //Calculates E^x, x is a number
```

```
print(s)  //Prints the value of s, for debugging, and returns it
```

```
sin(x)    //Calculates the sine of x, x is a number
```

```
sqrt(x)   //Calculates the square root of x, x is a number
```

```
log(base,value) //Calculates the logarithm on the given base  
                //of the given value,both numbers.
```

And as a final tool HulkI supports recursion, functions that call themselves.

An example of such function for calculating factorial:

```
function factorial(integer) => if(integer < 0) "Factorial not defined  
for negative numbers" else if(integer == 0) 1 else integer *  
Factorial(integer - 1);
```

```
> let have="a nice", sunny=" day" in have @ sunny;  
a nice day  
> And for you who readeed this, lets make an exception;  
! SYNTACTIC ERROR      Malformed expresion.  
In: And for you who readeed this, lets make an exception;  
      ^  
  
> "farewwel";  
farewwel  
> "THE END";  
THE END  
>
```