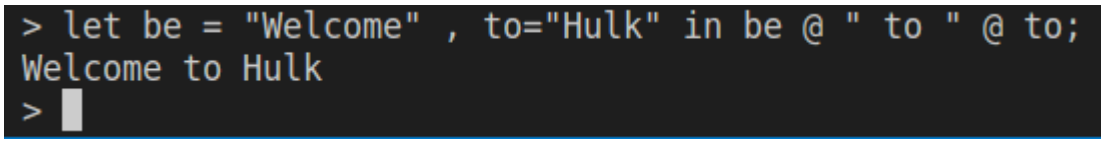
HulkI

A Hulk interpreter.

Documentation



What is this document about?

This document is the official documentation of the HulkI interpreter.

The HulkI interpreter is provided as source code on the source folder on the same folder as this document.

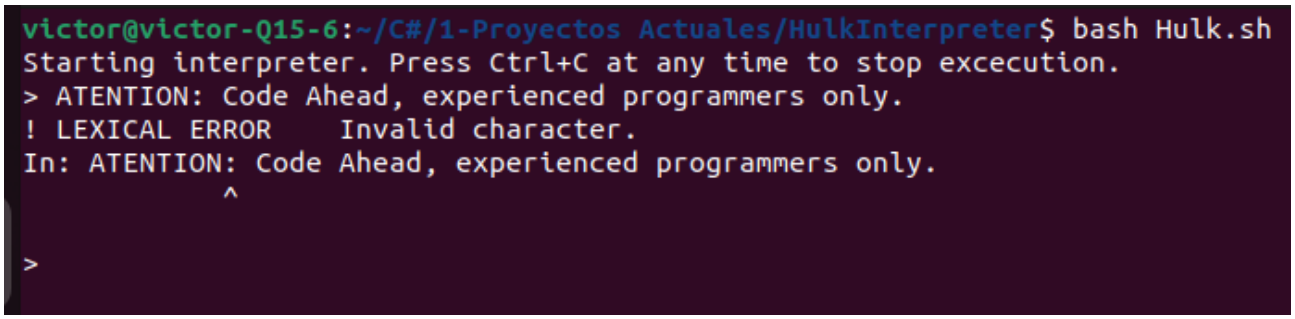
HulkI is the interpreter.

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

Guidance on how to setup HulkI, use the interpreter, and examples of

Hulk code are provided on the user manual, please refer to it if that is what you are looking for.

This document deals with implementation details of the HulkI interpreter.



The life cycle of a Hulk expression

This interpreter executes exclusively Hulk one-liners, which is I like to call Hulk one line expressions.

On start an interactive session is set up where the user input its code and the results are showed immediately.

Code inputted by the user is called an expression. Almost every expression has a return value, except function declarations.

The code is then scanned by the Scanner and a list of tokens is produced as a result. Tokens represents regions of source code with a well defined meaning. They are building blocks for the upcoming infrastructure.

The tokens are then supplied to the Parser which form expressions with them. Expressions build on top of tokens, they are a higher level representation of code, whit a tree-like structure, which better represents the meaning of the code. This tree structure is called the Abstract Syntax Tree , abbreviated AST. On HulkI the method used for producing the AST is the recursive descent parsing.

After the AST as been produced it is supplied to the Interpreter, the class, which evaluates then the result of an expression. This is done by recursion because expressions may depend upon other expressions until a simple expression, like a number, is found whose evaluation is simple enough. The result of the evaluation is then printed on the console and the process restart again.

Exceptions may arise during any of the previous steps, but as they are expected they are catched and error messages are printed so the user can fix the errors that lead to the exception.

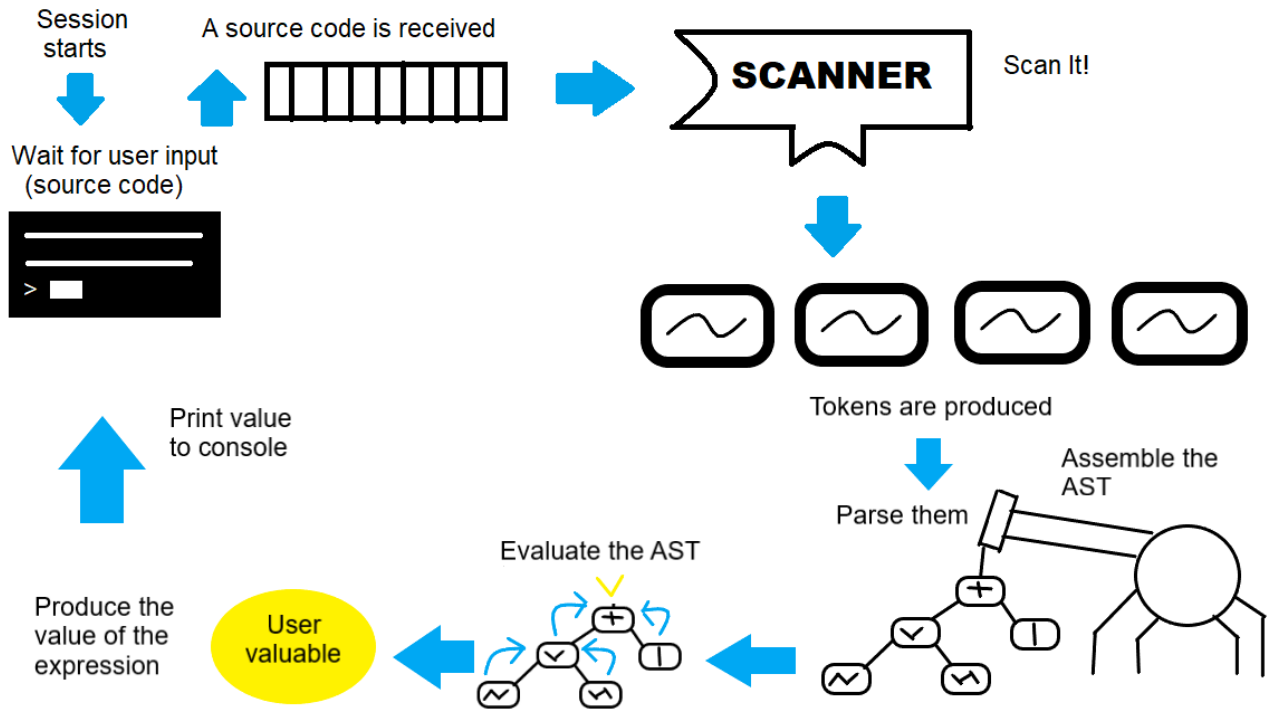


Figure 1: The process of executing expressions

The code of HulkI

On this section will be explained the classes that made the HulkI source code.

Class Hulk on file Hulk.cs

This class contains the Main method and thus its the entry point of the application.

It has an Interpreter instance as a member that will be reused through the the session, this allows to store previously declared functions for future usage.

On the Main method it offers support for testing mode and to error mode which are for testing purposes.

If no mode is specified via arguments to the program it enters REPL mode. On this mode it receives code one line at a time an immediately interprets it printing the value the expression evaluates to.

It maintains the last line of code in the lastLine member for error printing purposes.

The Run method receive the source code, a line. If the line is empty or only contains whitespace it does nothing. If the line is not empty it sets lastLine. Use a Scanner to scan the source code and return a list of tokens. Use a Parser to parse the list of tokens and obtain a Abstract Syntax Tree, on a Expr object. Then interprets the AST and return the result of the evaluation as a string object to be printed on console. Exceptions may arise during this process. Some are expected, those derived from HulkException, others are not. Both are catched, keeping the REPL session alive.

The Error method handles general error printing on the CLI. It prints the error type with a description of the error and optionally the line where the error occurred, usually lastLine, and an approximation to the location of the error.

The ScannerError, ParserError and InterpreterError share the same implementation and are called by their similar named HulkExcpetions.

The DebugScanner and DebugParser methods are not currently used but their purpose is to print the tokens produced by the scanner and the AST produced by the parser respectively. They where used on early development stages for debugging purposes.

Class AutomatedTesting on file AutomatedTesting.cs

This class is for testing the interpreter. After the Interpreter class was up and running this become the main debugging method for the interpreter.

The structure Test represents a test as a source code to be executed and a expected value from the interpreter.

The testing process is then resumed as follows : execute source, collect result, compare result and expected value, report.

The member tests is a Test[] which contains all the tests. Tests are added programatically to the source code. Its designed this way because at the moment it was simple and easy to implement it like this. Its also possible to put tests on a file, and make a third application that handles the logic of testing. This way is encouraged so testing code is separated from other code. This improvement will be made on future releases.

The Test method just wire up all the parts involved on testing. It loads the tests, feeds them to the interpreter one at time, collects the result from the interpreter, compare if the result is as expected and report accordingly. It handles the special case of function declarations where the result from the interpreter is the empty string.

A word on automated testing: its of great importance to do automated testing. After every change a run of testing was performed and this aided to address bugs instantly avoiding them to pile up and bring nasty consequences.

TokenType.cs file.

Token types are defined on this file using an enumeration.

The purpose token types is to differentiate tokens and thus the operations that should be performed on them and how.

As an example NUMBER tokens can be summed, because literal expressions build on top of them, but cant be called, which is a privilege of call expressions, whose main part is an IDENTIFIER token. Thus only NUMBER tokens can be summed and only IDENTIFIER tokens can be called.

Class Token on file Token.cs

This class represents tokens, the units produced by the scanner after processing the source code. They are foundational on this interpreter and are used since the beginning of the development process.

A Token contains an offset, which is the position of the first character of this token on the line it was found.

For example :

    3 + alfa;

The underlined position, 4, corresponds to the offset of the token that represents the identifier alfa.

A Token has a type, types can be viewed on the file TokenType.cs

A Token has a Lexeme, which is the portion of source code from which the token was generated.

A Token can have a Literal value, which can be null, associated to it.

Tokens also offer a ToString method for debugging purposes.

Tokens are simple but powerful.

Note that most tokens doesn't have a literal value, only those who represents string literals and numbers.

Note that for string literals their lexeme includes the quotes but their literal value does not.

Class Scanner on file Scanner.cs

The Scanner class receives Hulk source code and translate it into tokens.

The field source keeps the source code string.

The field tokens is the list of tokens produced from the source.

The field current gives the position on the source string of the next character to be processed.

The field start gives the position on the source string were the actual token started.

A visual example illustrating the meaning of the fields.

source "let a = "An example" in a @ "By Vex";"

current   ^

start   ^

tokens [let , identifier(a) , equal, string("An example")]

The keywords fields is a predefined dictionary that translates a string to a TokenType.

The constructor receives a source code and initializes source with it. The fields start and current are initialized as 0 by default and tokens is initialized as an empty list.

The Scan method keeps scanning tokens until it run out of characters. It sets start to current to update the start of the new token and calls the ScanToken method. It also catches exceptions that may arise during the scanning phase.

The ScanToken method handles one-char, two-char operators by itself, ignore any whitespace, tabulation and new line, and auxiliates on the ScanNumber, ScanIdentifier and ScanString methods.

What is an identifier ? An identifier is a piece of consecutive letters, digits and underscores where the first character must be a letter or an underscore.

It can be defined like this:

    ID = (LETTER | "_") (LETTER | DIGIT | "_")*;

This way the start of an identifier is different from the start of a number.

The ScanIdentifier method consumes every possible character that can be part of the identifier, also known as Maximal Munch principle. Notice that on this phase an identifier can also be a keyword, so if it matches a keyword the is treated as one.

Maximal Munch Principle:

Consider 'inPut'.

The correct way to tokenize it is by treating it as a whole identifier.

This other way is incorrect : [in , identifier(Put)]

So we must consume every character before deciding which kind of token it is.

What is a number ? A number is a piece of consecutive digits, optionally followed by a dot '.' and one or more digits.

It can be defined like this: NUMBER = DIGIT+ ("." DIGIT+)?

This representation ensures:

-At least one digit before the dot, DIGIT+ means at least one digit.

-Optional fractional part, offering support for both integers and real numbers. ("." DIGIT+)? means at most one dot followed by at least one digit.

-At least one digit after the dot, because of the DIGIT+ rule.

As a consequence of this rule leading zeroes are permitted but ignored.

For example:

000001;

The value of the above expression will be 1.

The ScanNumber method :

-Scans the leading digits.

-If after that exist a dot and a digit after the dot it scans the dot and all the trailing digits.

What is a string? A string is all the text between two quotes, so the quotes are necessary and the text is optional.

It can be defined like this :

STRING = '"' CHARACTER* '"';

Strings also offer support for the escaped characters: '\t' for representing tabs, '\n' for representing newlines and '\"' for representing quotes.

The ScanString method consumes all the characters until it hits a quote, which is treated as an enclosing quote or hits the end of the source, which is an error. It also avoids treating the quote on a '\"' as the enclosing quote.

The string literal value is all the text between the quotes without the quotes.

After computing this value, all escaped characters are replaced by the characters they represent and a STRING token is created.

The ScanToken, ScanIdentifier, ScanNumber and ScanString methods auxiliate on the AddToken method.

The AddToken method adds a token to the list of tokens. It computes the lexeme of the token based on its start position and the current position.

The IsAtEnd method indicates whetter the end of the source has been found or not.

The Advance method returns the character at position current and moves current one position ahead. This is called consume a character.

The Peek method returns the character at position current but doesn't move current.

The Match method returns true whetter the given character is the same as the character at current position and moves current one position ahead. Its like a conditional Advance.

The IsDigit method returns true when the given character is a digit.

The IsAlpha method returns true when the given character is a letter or an underscore.

The IsAlphaNumeric method returns true when the given character IsDigit or IsAlpha.

Expressions.cs file

On this file several classes are declared, they represent the AST nodes.

Class Expr:

This class is the base class for all the other expression classes. The generic method Accept its intended to apply the visitor design pattern. All derived classes from Expr implements the Accept method like this:

```
public override R Accept<R>(Visitor<R> visitor){
   return visitor.Visit$CLASS_NAME$(this);
}
```

Interface Visitor<R>:

This interface is designed to work in conjunction with the Accept method from the Expr class. First of all a Visitor should be a class that handle expressions of multiple types and behaves differently according to the type of the expression . Example of visitors are:

-The AstPrinter : because printing depends on the type of expression beign printed, and a BinaryExpr is printed differently from a LetInExpr.

-The Interpreter : because different expressions are interpreted differently.

-The Replacer : because replacement is done differently according to the type of expression, a LiteralExpr requires no replacement at all while a VariableExpr its the main target for a replacement.

By adding a new Visit method you can support a new type of expression, this way all the current visitors must offer support for this kind of expression to.

The Interpreter, AstPrinter and Replacer are visitors and implement all the visit methods but they do it differently and they encapsulate their implementations. so all the code for interpreting expressions is contained in the Interpreter class. A visitor can receive an Expr and call

its Accept method which will call back the proper visit method on the visitor.

Class LiteralExpr:

This class represents number literals, string literals, true and false and the E and PI constants.

Class UnaryExpr:

This class represents unary operations like:

! Logical negation

- Opposite number.

Class BinaryExpr:

This class represents binary operators like:

+ Sum

- Subtraction

* Multiplication

% Modulus

/ Division

^ Exponentiation

@ String concatenation

Comparison operators:

< Less than

<= Less than or equal to

> Greater than

>= Greater than or equal to

== Equal to

!= Different from

Logical operators:

& Logical and

| Logical or

Class ConditionalExpr:

This class represents if-else expressions.

It has fields for the condition expression, the if branch expression (executed if condition is true) and the else branch expression executed if the condition is false).

It also has two additional properties for error printing purposes:

IfOffset and ElseOffset.

Class LetInExpr:

This class represents a let-in expression.

It has fields for the assignments and the in branch expression (expression after the 'in').

The assignments are stored as a list of AssignmentExpr.

Class AssignmentExpr:

This class represents an assignment.

It has fields for the identifier and the rvalue expression.

Class VariableExpr:

This class represents variables.

It has a field for the IDENTIFIER token that represents the variable.

Class FunctionExpr:

This class represents function declarations.

It has fields for the functions name, body and arguments, and provides a propertie to determine the arity of the function.

It also has a RandomizeArgs method that work in conjunction with the Register method from Environment. To each argument it appends a salt and in the body of the function every occurrence of the old argument its replaced by the new salted version. This replacement is done by the Replacer class.

Class CallExpr:

This class represents a function call.

It has fields for the name of the function begin called and the parameters passed.

It also has a propertie to retrieve the arity of the call, the amount of parameters passed.

Class Parser on file Parser.cs

This class creates an AST from a list of tokens.

The field tokens is the list of tokens to be parsed.

The field current is the position on tokens of the next token to be parsed.

List of tokens:

```
Number  Plus  Number Semicolon EOF
  2             3
                ^
              current = 2

_____________ ___________________________
   Parsed          To be Parsed
```

The current field is set to 0 by default and tokens is initialized on the constructor. This way the Parser feeds on the output from the interpreter.

The Parse method its the entry point of the Parser. It starts the recursive descending parsing by calling the HLExpression method. Please refer to the Grammar.txt file to visualize the recursive descending structure who follows the parser. It also capture errors occurred during the parsing and calls the HandleException method, a similar pattern is followed on the interpreter and scanner for error handling. On this method some errors are detected like an expression that doesn't ends on a ';'.

The HLExpression method parses a function declaration, with all its requirements, if it founds the 'function' keyword, else it falls to Expression (falls to means it descends). The above rule guarantees that no function declaration can be inside of any kind of expression. So function declarations are special types of expressions.

The Arguments method represents the Params rule of the grammar, it just captures the arguments. It offers assistance to the HLExpression method for parsing function declarations.

The Expression method engulfs every other Hulk expression. It falls to Declaration.

The Declaration method parses let-in expressions or falls to Conditional. Here is seen by the first time the recursive nature of this calls since for parsing the expression after the 'in' keyword it calls the Expression method. Also this is an example of right associativity, although indirectly, because the expression after the 'in' can be a let-in expression.

The Assignment method helps the Declaration method by parsing the variable declarations between the 'let' and the 'in'. It treats no assignments like an error.

The Conditional method parses if-else expressions or falls to Or. It also calls the Expression method for parsing the expression on the condition, the if branch and the else branch.

The Or method parses an arbitrary number of operands related by the '|' operator, for logical or, from left to right; because its left associative; or falls to And.

'w | x | y | z' its parsed as: ((w | x) | y) | z

The And method parses an arbitrary number of operands related by the '&' operator, for logical and, from left to right; because its left associative; or falls to Equality.

'w & x & y & z' its parsed as: ((w & x) & y) & z

The Equality method parses an arbitrary number of operands related by the '==' or '!=' operators, but it stops if it found more than one because that its not supported and reports it as an error.

a == b is parsed as usual.

a != b is parsed as usual.

a == b == c is an error.

a == b != c is an error.

a != b == c is an error.

a != b != c is an error.

However if parenthesis are added it works,from a syntax perspective:

(a == b) != c , OK

a == (b != c) , OK

The Comparison method parses an arbitrary number of operands related by the '>','>=','<' or'<=' operators, but it stops if it founds more than one because that its not supported and reports it as an error. Just like the Equality method.

a > b is parsed as usual.

a > b > c is an error.

a < b <= c is also an error.

This should be expressed as:

a < b & b <= c

On the Equality method and the Comparison method an invalid syntax is allowed to be parsed because that way it can be reported as an error easily.

The Term method parses an arbitrary number of operands related by any combination of the operators '+','-' and '@', from left to right because they are left associative, or falls to Factor. Note that later in the interpreter errors might be reported by using arithmetic operators and the string concatenation operator on the same expression due to invalid operands, ie: an '@' who receives a number as his right operand.

2 + 3 + 4 is parsed as (2 + 3) + 4

2 - 3 - 4 is parsed as (2 - 3) - 4

2 + 3 - 4 + 5 - 6 - 7 is parsed as (((((2 + 3) - 4) + 5) - 6) - 7)

The Factor method parses an arbitrary number of operands related by any combination of the operators '*', '%' and '/', from left to right because they are left associative, or falls to power.

2 * 3 * 4 is parsed as (2 * 3) * 4

2 / 3 / 4 is parsed as (2 / 3) / 4

2 % 3 % 4 is parsed as (2 % 3) % 4

2 % 3 / 2 * 4 is parsed as ((2 % 3) / 2) * 4

The Power method parses an arbitrary number of operands related by the '^' operator, from right to left because it is right associative, or falls to Unary.

5 ^ 4 ^ 1 is parsed as 5 ^ (4 ^ 1)

The Grouping method parses a parenthesized expression, or falls to Call if no parenthesis are found.

The Call method parses a function call. It parses its parameters through the auxiliary function Parameters.

The Parameters method parses a comma separated list of parameters of arbitrary size. The parameters it parses are expressions. It returns the list to the Call method to build the CallExpr.

The Literal method parses the following expressions:

-string literals

-number literals

-true keyword

-false keyword

-PI constant

-E constant, for Euler.

-Identifiers, which do not constitute calls, but variables.

If the token doesn't fall in any of this categories it does call the helper method Unrecognized.

The Unrecognized method just throws exceptions. Its reached when an expression is expected but no suitable one is found. It also detects when function declarations are being mixed with other expressions.

The Peek propertie returns the current token.

The PeekNext propertie returns the token one position ahead of the current token. This is called a look-ahead of one.

The Advance method returns the current token and moves current one position forward. This is called consume the token.

The Match method checks if the type of the current token , if it is of one expected type it consumes the token and returns true, if not it returns false. Its like a conditional Advance.

The Consume method , given a message , an offset and a list of types, checks if the current token is of any of the types and if it is not it throws an exception with the given message and offset. Its a replacement for :

```
if(!Match(...)) throw new ParserException(message,offset)
```

The Previous propertie returns the token one position behind current.

The HasSemicolon propertie returns true if a semicolon exist on the remaining tokens, current included.

The Offset propertie returns the offset of the current token.

Class AstPrinter on file AstPrinter.cs

This class is for printing the expressions produced by the parser, so the parser output can be viewed and thus facilitating debugging. It was more useful in early stages of development.

Its entry point is the Print method which receives an expression. From there the corresponding Visit method is indirectly called.

Each of the Visit methods handle the printing of specific types of expressions.

Class Interpreter on file Interpreter.cs

This class interprets, executes, the source code and produce the results.

It has an environment field where variables and functions are stored, and who provides some other functionalities.

It has a NestedCallCount field which is the amount of nested function calls the code has done. This is reseted to 0 every time a new line of code is to be interpreted.

It has a MaxNestedCallCount field which is the maximum amount of nested function calls that a code can do. Its used to avoid stack overflow exceptions, which cannot be captured during runtime and cause the end of REPL session resulting in a bad user experience. However the process of capturing a stack overflow is slow, due to unrolling the calls.

The Interpret method is the entry point of the Interpreter, it receives the AST and returns a string version of the result or null if it interprets a function declaration. It also resets NestedCallCount to 0 on every new call. On this method exceptions are cached and rethrowed by calling the HandleException method, a scheme reused on the scanner and the parser entry points.

The Evaluate method is where the visitor pattern starts working, it receives an Expr and call its Accept method which will call back the correct visit method from the interpreter.

Methods on the different types of expressions:

VisitLiteralExpr:

A literal expresion evaluates to its value.

VisitUnaryExpr:

Operator ! works only with a boolean operand. Returns its negation.

Operator - works only with numbers. Returns its opposite.

Note: Works only with 'x type' implies that an error is throwed if a different type is provided.

VisitBinaryExpr:

Evaluation of binary expressions develops as follows. Evaluate the left side expression. Evaluate the right side expression. Then evaluate the operation.

Operator + works only with numbers. Return its sum.

Operator - works only with numbers. Return its difference.

Operator @ works only with strings. Concatenate the right string to the left one and returns the concatenation.

Operator * works only with numbers. Return its product.

Operator % works only with numbers. Return the rest of the division of the left number by the right number.

Operator / works only with numbers. Return the quotient of the division of the left number by the right number.

Operator ^ works only with numbers. Return the result of raising the left number to the right number.

Operators < <= > >= work only with numbers. Work as usual. Returns a boolean value.

Operators != == use the auxiliary method IsEqual. Returns a boolean value.

Operator & works only whit booleans. Does not shortcircuit. Represents Logical And. Returns a boolean value.

Operator | works only whit booleans. Does not shortcircuit. Represents Logical Or. Returns a boolean value.

VisitConditionalExpr:

A conditional expression first evaluate its condition. If its true then return the result from evaluating the 'if' branch expression. If not then return the result from evaluating the else branch expression. In the case that the condition does not evaluates to boolean its an error.

VisitLetInExpr:

In a let-in expression the 'in' expression can access to the variables declared after the 'let' and before the 'in', and those variables dissapear after that.

To achieve this behavior it uses methods from the Environment class. First the variables are setted with their corresponding values. To do this the rvalues of the assignments are evaluated and through the Set method from environment those values are binded to the variables. Variables ready. Now the expression after the 'in' is evaluated and its value stored for returning it later. After that the variables must dissapear, because they go out of scope. This is achieved by calling the Remove method from the Environment class and providing it with the variable identifier. Now the stored value its returned.

Its worth remembering that the interpreter owns a instance of Environment called ,creatively, environment which is a big global mutable Environment.

VisitAssignmentExpr:

AssignmentExpr are not evaluated as they only exist within the LetInExpr and the VisitLetInExpr handles them.

VisitVariableExpr:

Evaluating a variable is returning its associated value. This is done via the Environment Get method passing the identifier of the variable.

VisitFunctionExpr:

FunctionExpr doesn't evaluate to anything so this returns null. Before returning the function is registered via the Environment Register method passing the FunctionExpr as parameter.

VisitCallExpr:

This is where functions calls do their magic. First it checks if its in presence of a builtin function, if so it retrieve its parameters and evaluates them, because they are expressions. Then identify which builtin is to be called, check the parameters and evaluate that function returning its result. If its not a builtin function it checks if exists a function with the given name, if not its an error. After that it checks if

exists a function with this name and arity, if not is an error. If the name and arity are correct, they represent a declared function, it continues.

A LetInExpr is used for evaluating function calls based on the following observation:

-A function argument and a function parameter can be used to make an AssignmentExpr where the argument provides the variable and the parameter provides the rvalue.

-After this initial setup the body of the function can be executed like the expression after the 'in' on a let-in expression.

So evaluating a function call from this point its creating a LetInExpr based on the previous observations, and the result from evaluating that LetInExpr is the result of the call. This is code reusability.

The helper method IsEqual is used for checking equality, it uses the Equals method from C# and it returns true wheter the objects being compared have the same value because they are, numbers(floats), strings and booleans.

The family of helper methods Check$TYPE$Operand where TYPE is one of: Number, String or Bool behaves mostly in a similar fashion. They receive the token that represents the operation for error printing purposes. They receive the object they must check its type and an optional parameter pos, which indicates the position of the object respect the operator: -1 left, 1 right, 0 by default. Then they check the object is the expected type and if not they throw a fancy exception. The SetPosString and GetType methods are meant to help those previous checks to print its error messages.

Builtin functions are :

rand() generates a random number in the interval [0,1)

cos(x), exp(x), sin(x), sqrt(x) , similar to their C# equivalent.

log(base,value) for logarithm calculation.

print(expression) for printing. Prints on the console the value it contains and returns it. This is done to aid debugging process.

Class Environment on file Environment.cs

This class represents the state of the interpreter. There is just one big global mutable environment. Here lives functions and variables. Functions binds identifiers and expressions making expressions reusable. Variables binds identifiers to data, making data reusable.

Variables are stored on a dictionary. To each variable there is associated a list of objects that behaves like a stack. This way only the last object can be accessed simulating variable scope.

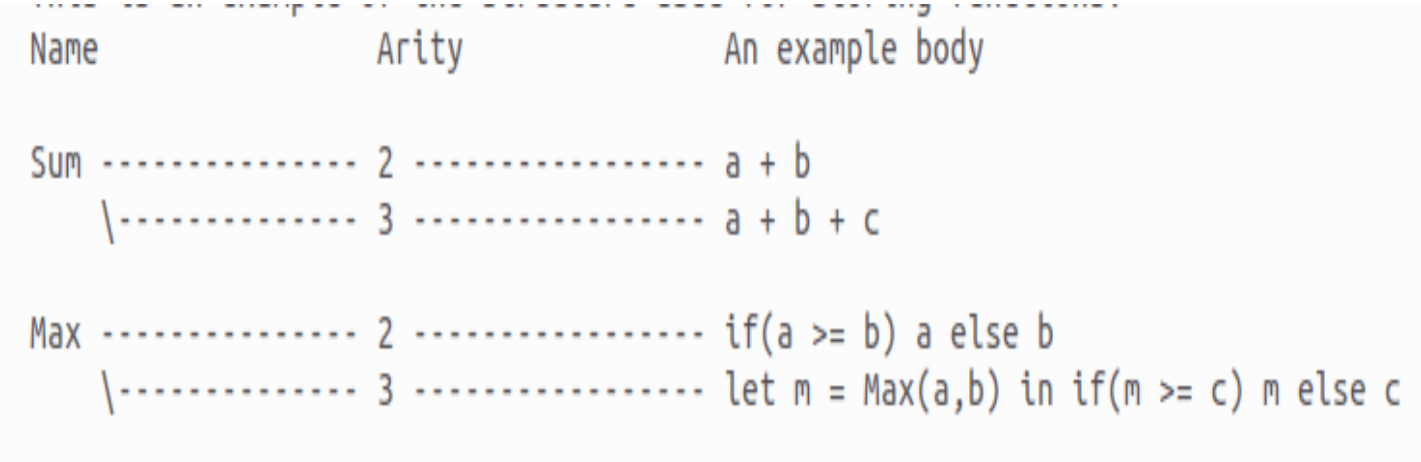
The Get method enforce this behavior. First if the identifier is from a function it report an error because functions cant be used as variables. If not it returns the last object of the list associated to the identifier, the top of the stack. If this identifier represents no variable its an error to access it.

The Set method is where variables are binded to values. If the identifier correspond to a function its an error to assign it a value. This way a identifier can only be a variable or a function but no both. A new dictionary entry is created for this identifier if its new, then the value is added to the end of the list of values the identifier represents, the top of the stack.

The Remove method removes the last value of the list associated to the given identifier, pops the stack. If the list becomes empty after that, the identifier gets removed from the dictionary, it gets out of scope.

The above three methods receive the token that represents the identifier because the token holds information important to error reporting.

Functions are stored on a dictionary. The dictionary structure is peculiar, as a function is uniquely represented by its name, its arity, and the type of its arguments. The last one is not used in the current implementation.



This structure can be extended to support types because types comes after arity in the hierarchy.

The Register function associates an identifier to a function. It handles function declarations. It forbids redeclaration of functions. Note that you can declare a function :

function cos(alfa,beta) => "some body";

because the arity does not match the builtin cos(x) for cosine calculation.

After this the arguments are randomized, salted specifically because a number is added to the end.

Consider the following snippet:

function fun(sum) => "do something";

fun(2); //prints "do something"

function sum(x,y) => x + y;

fun(2); //It should print "do something" despite its argument is as the name of a function.

By salting the arguments you can : have the arguments be named like functions, which of not being possible would be conflictive since we cant tell which names would be used to declare functions, so we avoid future problems. The way its done its explained in the Replacer class. Next is declaring the function if it doesn't exist, or reporting an error if it exist because redeclaration is not allowed. A function exist if its name and arity match an existing function name's and arity's.

The IsFunction method works over the function dictionary structure.

The methods GetBody and GetArguments allow access to those specific elements for the provided function.

The method GetAritys provides all the diferent arities of the provided function name.

The method IsBuiltin determines if a given function is a builtin function.

The member builtIns declare the name and aritys of the builtin functions.

Class Replacer on file Replacer.cs

The Replacer class replaces all occurrences of an identifier by another identifier on an AST. This helps addressing the following situation:

A function is declared:

```
function thisIsFun(sum) => "ja ja ja";
```

I can call it

```
thisIsFun(2);
```

But if i declare sum to be a function

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

And i call thisIsFun again

```
thisIsFun(2);
```

This is an error because the argument sum also collide with a function name.

Thus replacing is a technique to avoid that arguments names collide with function names.

The Replacer class implements the Visitor interface in order to process each kind of expression.

The field viejo contains the old identifier, the one we want to replace.

The field nuevo contains the new identifier, the one we want to use as a replacement.

The constructor sets the fields viejo y nuevo and calls the Replace method with the provided AST.

An example to illustrate how Replacer works:

OLD_CODE:

```
let a = 12 in 3 ^ a * (5 + a);
```

We want to replace a by b:

NEW_CODE:

```
let b = 12 in 3 ^ b * (5 + b);
```

The Replacer class just do that.

Notice that it received an expression and returned an expression, that is why it implements Visitor<Expr>. Also notice that the Replace method is doing a Pre-Order traversal of the AST because it visits all the children of a node(the nested expressions on an expression), before computing the new expression.

A LiteralExpr doesn't need replacements because it has no identifiers.

An UnaryExpr can have identifiers on the operand, so the operand is recursively replaced.

A BinaryExpr can have identifiers on any operand, so both operands are recursively replaced.

A ConditionalExpr can have identifiers on the condition, the if-branch expression and the else-branch expression, so those are recursively replaced.

A LetInExpr can have identifiers on the assignments and the in-branch, so those are recursively replaced.

An AssignmentExpr can have identifiers as the variable being assigned to as well on the rValue expression. The rValue expression is recursively replaced, the identifier of the variable being assigned to is replaced instantly in case of a Match.

A VariableExpr is an identifier, so if it Match its replaced.

A FunctionExpr can have identifiers as parameters and on the body expression, so the parameters are replaced if needed and the body is recursively replaced. Notice that the main target of the Replacer are FunctionExpr.

A CallExpr has an identifier for the functions name and can have identifiers on the parameters. The functions name is replaced if needed and each parameter, which is an expression, is recursively replaced.

The Match function receives a Token, for the identifier, and if it is different from viejo it returns the same Token, but if its the same as viejo it returns a copy of nuevo with the offset of the given identifier.

File Error.cs

On this file exceptions are defined for specific stages of the lifecycle of the program.

Class HulkException:

This class is the base class for all Hulk defined exceptions. It stablish an Offset propertie, which is the position in the source code where the error was found. Its an approximation for error printing purposes. It also stablish an ErrorType method which returns one of the following three strings:

Lexical, Syntactic, Semantic.

This are the three kinds of errors that can occur during the execution of a Hulk program.

Lexical errors happens during the scanning phase. In the Scanner class.

Syntactic errors happens during the parsing phase. In the Parser class.

Semantic errors happens during the execution phase. In the Interpreter class.

The HandleException method provides uniformity for handling errors in the Scanner, Parser and Interpreter class. Its implemented like this by all three classes.

```
public override void HandleException()
{
    Hulk.Error(this);
    throw this;
}
```

Classes ScannerException, ParserException, InterpreterException :

This classes represents Lexical,Syntactic and Semantic errors respectively and as their names suggest they are encountered at the Scanner,Parser and Interpreter class respectively and during the scanning, parsing, interpreting phase.

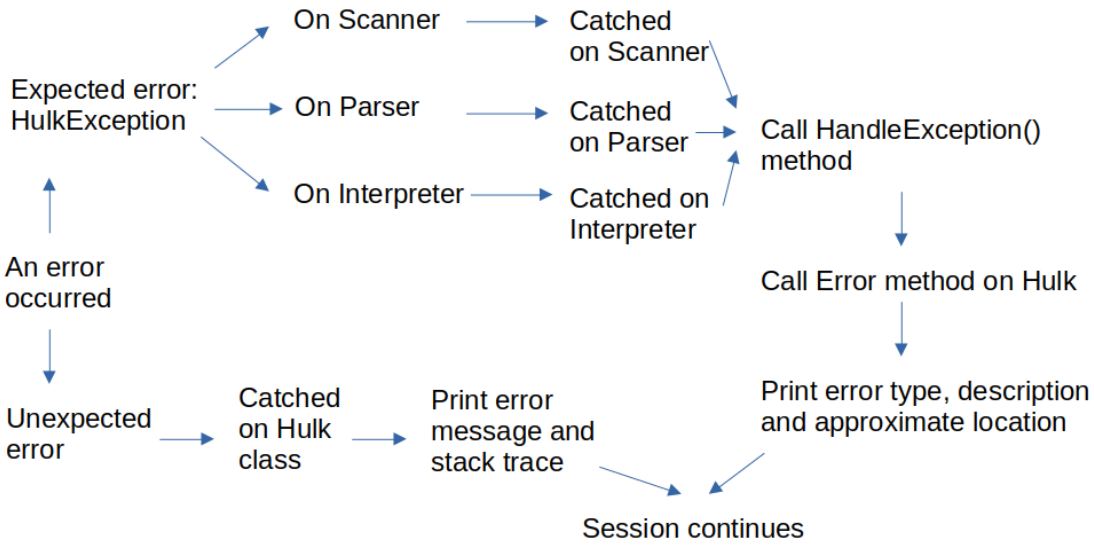


Figure 2: Error Handling Scheme

The End