HULK

Rafael A. Sanchez Martinez
University of Havana
Group: C-113

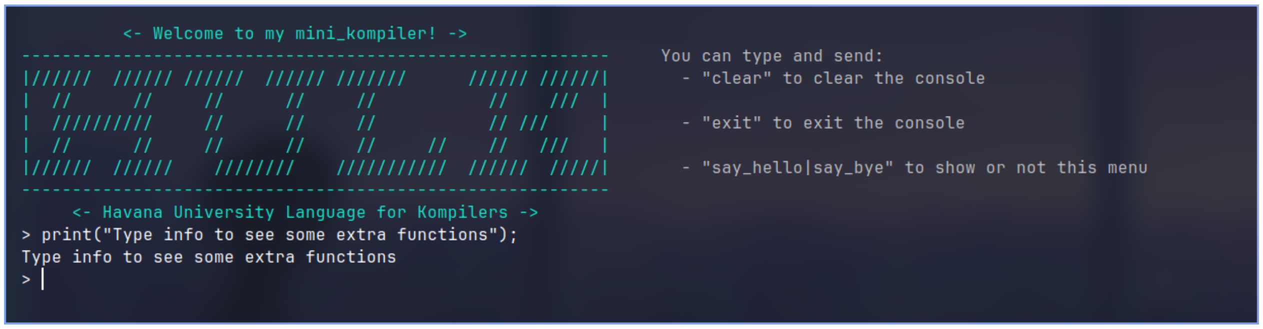
Figure 1: Havana University Language for Kompilers

1 Introduction

Welcome to the user manual of HULK, the programming language of the University of Havana. This manual is divided into 5 sections, each one dedicated to a part of the interpreter. The first section explains how to install the interpreter, the second section explains how to use the interpreter, the third section explains how to use the lexical analyzer, the fourth section explains how to use the syntax analyzer and the fifth section Explains how to use the expression evaluator.

1.1 Start

- To start using the interpreter, you must download the source code of the interpreter from the project's github repository. Once the source code has been downloaded, the project must be run using the 'dotnet run' command from a terminal open in the project folder.
- Another option is to run the script which will take care of everything
- See previous report in `https://github.com/ARKye03/mini_kompiler.git`



```
         <- Welcome to my mini_kompiler! ->
-----------------------------------------------------------
|//////  ////// //////  ////// ///////      ////// //////|
|  //      //     //      //     //           //    ///  |
|  //////////     //      //     //           // ///     |
|  //      //     //      //     //     //    //   ///   |
|//////  //////    ////////    ///////////  //////  /////|
-----------------------------------------------------------
     <- Havana University Language for Kompilers ->
> print("Type info to see some extra functions");
Type info to see some extra functions
> |
```

```
You can type and send:
  - "clear" to clear the console

  - "exit" to exit the console

  - "say_hello|say_bye" to show or not this menu
```

Figure 2: Havana University Language for Kompilers

1.2 Summary

HULK (Havana University Language for Compilers) is an educational, type-safe, object-oriented and incremental programming language, designed for the Introduction to Compilers course of the Computer Science degree at the University of Havana.

A simple 'Hola Mundo' in HULK looks like this:

print('Hola mundo')

From a bird's eye view, HULK is an object-oriented programming language, with simple inheritance, polymorphism, and class-level encapsulation. Additionally, in HULK it is possible to define global functions outside the scope of all classes. It is also possible to define a single global expression that constitutes the entry point to the program.

Most syntactic constructs in HULK are expressions, including conditional statements and loops. HULK is a statically typed language with optional type inference, meaning that some (or all) parts of a program can be annotated with types and the compiler will check all operations for consistency.

But, in this case, since it is a project for 1st year students, it is a bit simplified. In this case, let's say, we have a one-line interpreter, which allow us, to evaluate expressions like:

- Like I said above, print("Hello World")
  - Which is equivalent to $print(``Hello'' + ``World'')$
  - And print("Hello" @ " World")
- All basics 'instructions' can handle simple math expressions like:
  - $``2 \pm 5;''$ —— $``147.6 \pm -63;''$
  - $``8 \times 2;''$ —— $``200 \div 3;''$
  - $``2^{64};''$ —— $``sqrt(3969)'' \rightarrow ``\sqrt{3969}''$ —— $``\log(2, 10);''$

- Mathematical functions:

Sin(x):	Returns the sine of x, where x is in radians. Example: Sin(0) returns 0.
Cos(x):	Returns the cosine of x, where x is in radians. Example: Cos(0) returns 1.
Tan(x):	Returns the tangent of x, where x is in radians. Example: Tan(0) returns 0.
Log(x):	Returns the natural logarithm (base 10) of x. Example: Log(1) returns 0.
Ln(x):	Returns the natural logarithm (base e) of x. Example: Ln(1) returns 0.
Sqrt(x):	Returns the square root of x. Example: Sqrt(4) returns 2.
Abs(x):	Returns the absolute value of x. Example: Abs(-5) returns 5.
Pow(x, y):	Returns x raised to the power of y. Example: Pow(2, 3) returns 8.
Exp(x):	Returns e raised to the power of x. Example: Exp(1) returns approximately 2.71828.
Floor(x):	Returns the largest integer less than or equal to x. Example: Floor(1.5) returns 1.
Ceil(x):	Returns the smallest integer greater than or equal to x. Example: Ceil(1.5) returns 2.
Round(x):	Rounds x to the nearest integer. Example: Round(1.5) returns 2.
Rand(min, max):	Returns a random integer between min (inclusive) and max (exclusive). Example: Rand(10, 20) returns a random number between 10 and 20.
Factorial(x):	Returns the factorial of x. Example: Factorial(5) returns 120.
Fibonacci(x):	Returns the xth number in the Fibonacci sequence. Example: Fibonacci(5) returns 5.
IsPrime(x):	Returns true if x is a prime number, false otherwise. Example: IsPrime(5) returns true.
IsEven(x):	Returns true if x is an even number, false otherwise. Example: IsEven(5) returns false.
IsDivisible(x, y):	Returns true if x is divisible by y, false otherwise. Example: IsDivisible(10, 5) returns true.
IsPalindrome(x):	Returns true if x is a palindrome, false otherwise. Example: IsPalindrome("radar") returns true.
Max(x, y):	Returns the maximum of x and y. Example: Max(2, 3) returns 3.
Min(x, y):	Returns the minimum of x and y. Example: Min(2, 3) returns 2.

- And thats pretty much it $\wedge \times \wedge$

• Evaluable expressions are:

  - Printing:

    This expression keyword receives another expressions as argument, and evaluates it, and print the returned value

    $print(55)$; shows 55 —— $print(\text{“}Kitty'')$; shows Kitty

  - Variables:

    This one, contains two important keywords, “LetKeyword” and “InKeyword”.

    A basic let-in expression is: “let $< var\_name >$ in $< statement >$”

    Example: “let x = $(63 \pm 109)^2$ in $print(x)$; ”

  - Conditions:

    Basic and classic conditions:

    if (condition) $< do\_if\_true >$ else $< do\_if\_false >$

    “if (1024 % 2 == 0) $print(\text{“}Even'')$ else $print(\text{“}Odd'')$; ”

  - Functions:

    ∗ Declare functions:

      To declare functions simple do:

      function $function\_name$(arguments) => $< statement >$;

      Example: function Pow(x, y) => x^y;

      This can be used like:

      “let number = $Pow(2, 5)$ in $print(number)$; ”

  - Note: A statement is basically another instruction or expression.

- There are also some predefined constants:

CoPI:	Arquimedes's constant. $\Rightarrow$ 3.14159.
CoE:	Euler's constant. $\Rightarrow$ 2.71828.
CoPyC:	Pythagoras's constant. $\Rightarrow$ 1.41421.
CoThC:	Theodorus's constant. $\Rightarrow$ 1.73205.
CoG:	Newton's constant. $\Rightarrow$ 6.6743.
CoPhi:	Aurea's constant. $\Rightarrow$ 1.61803.
CoGamma:	Euler-Mascheroni constant. $\Rightarrow$ 0.577216.
CoGc:	Catalan's constant. $\Rightarrow$ 0.915966.
CoK:	Khinchin's constant. $\Rightarrow$ 2.68545.
CoOmega:	Omega constant. $\Rightarrow$ 0.567143.
CoA:	Glaisher-Kinkelin constant. $\Rightarrow$ 1.28243.
CoM:	Mertens constant. $\Rightarrow$ 0.261497.
CoKp:	Kaprekar constant. $\Rightarrow$ 0.275823.
CoH:	Planck constant. $\Rightarrow$ 6.62607.

- Some expressions can be:
  1. print("Hello World");
  2. print$((((1+2)^3)\ *\ 4)\ /\ 5)$;
  3. print$(\sin{(2*PI)}^2\ +\ \cos(3*PI/\log(4,64)))$;
  4. function $\tan{(x)}$ => $\sin{(x)}\ /\ \cos{(x)}$;
  5. let x = PI/2 in $print(\tan{(x)})$;
  6. let number = 42, text = "The meaning of life is" in print(text @ number);
     - let number = 42 in (let text = "The meaning of life is" in (print(*text* @ *number*)));
  7. print$(7+(let\ x=2\ in\ x*x))$;
  8. let a = 42 in if (a % 2 == 0) print("*Even*″) else print("*odd*″);
  9. let a = 42 in print(*if* (*a* % 2 == 0) "even" else "odd");
  10. function fib(n) => if (n > 1) fib($n-1$) + fib($n-2$) else 1;

- And on and on, use your imagination, except if you are my programming teacher, please don't get creative, I beg you.

2 Lexer

2.1 Lexer Summary

A Lexer, also known as a lexical analyzer, tokenizer, or scanner, is a program or function that breaks down input code into a sequence of tokens. These tokens are meaningful units of code.

In the process of parsing programming languages, this task is traditionally split up into two phases: the lexing stage and the parsing stage. The lexer's job is simpler than the parser's. It turns the meaningless string into a flat list of things like "number literal", "string literal", "identifier", or "operator". It can also recognize reserved identifiers ("keywords") and discard whitespace.

The output of the lexer is used as the input for the parser. The parser then has the much harder job of turning the stream of "tokens" produced by the lexer into a parse tree representing the structure of the parsed language. This separation allows the lexer to do its job well and for the parser to work on a simpler, more meaningful input than the raw text.

More info about:

- What's a Lexer
- The Lexer - ACCU
- Extra blog

2.2 Now, my Lexer

First I've defined some simple Token types in this "enum":

```
public enum TokenType
{
    FLinq, //Link token for function declaration
    ComparisonOperator,
    Number, // Number
    StringLiteral, // Literally a string
    FunctionDeclaration, // Well, deprecated
    LetKeyword,
    IfKeyword,
    ElseKeyword,
    PrintKeyword,
    InKeyword,
    FunctionKeyword,
    Operator, // '+','-','*','/','^', '='...
    Punctuation, // '(',')' & ','
    Identifier, //Used for variables name and functions value
    EOL, //';'
    EOF, //Not yet
    Semicolon, //Idk man/woman
    Separator // '@', and in a future, "||", "&&"...
}
```

Basically, my Tokens, have a few properties:

```
public TokenType type; // From enum
public string value; // Text of the token
public int line; // Line position, currently 1, always
public int column; // Column position
```

`Note:` ToString override is there to test tokens only.

Some functions of my lexer

- advance(): One char to the right.
- skip_whitespace(): Jumps to next non-blank char.
- peek(): Takes a look to next token, without taking it.
- unget_token(): Drops next token, to use it later.
- Some Token functions. Take a summary along the code itself

The main function

The get_next_token() not that simple C# method that tokenizes a source code string for a programming language interpreter. This method reads characters from the source code and categorizes them into different types of tokens like operators, punctuation, keywords, etc.

If you're asking for an explanation of this code, here's a brief overview:

- The 'get_next_token' method is used to get the next token from the source code.
- It first checks if there are any tokens already read and not processed. If so, it returns the first one and removes it from the list.
- If there are no pre-read tokens, it starts reading the source code character by character.
- Depending on the current character, it categorizes it into a token type (like operator, punctuation, keyword, etc.) and returns a new token of that type.
- It also handles multi-character tokens like ' => ', ' == ', ' <= ', ' >= ', and '! = '.
- If it encounters a newline character ('
  n'), it increments the line count and resets the column count.
- If it encounters an unrecognized character, it prints an error message.
- If it reaches the end of the source code, it returns an EOF (end-of-file) token.

The 'function_declaration()': used to parse a function declaration in the source code of a programming language. Here's a brief overview of what it does:

- It first checks if the next token is a function keyword and then expects an identifier (the function name).
- It reads the function name by appending all the non-whitespace characters.
- It then checks for an opening parenthesis '(' to start the function parameters list.
- It reads all the parameters of the function, separated by commas ',', and adds them to a list.
- It then expects a ' => ' token, which presumably indicates the start of the function body.
- It reads all the tokens of the function body until it encounters a semicolon ';' or an EOF (end-of-file) token, indicating the end of the function body.
- Finally, it returns a 'FunctionToken' that represents the function declaration, including its name, parameters, and body. Note: This 'Token', turns to 'DFunction'.

3 Interpreter

3.1 Parser

A parser is a program or a component of a program that analyzes a string of symbols according to a set of rules. A parser can be used for different purposes, such as understanding natural language, processing computer languages, or parsing data structures. A parser typically breaks down the input into smaller units called tokens, and then assigns them to categories based on their syntax and semantics. A parser may also produce a representation of the input, such as a parse tree or an abstract syntax tree, that shows the hierarchical structure and the relationships between the tokens.

3.2 My Parser

In my case, I handle, 2 types of 'expressions', basic statements, and return expressions, the first ones, generally are used to print values, second ones to handle complex operations, you won't even notice them, there are a third type of expression, that derives from the return expressions, used only on declared functions.

My parsing life starts with the Run() function, used to parse and execute a series of statements in the source code of a programming language. This is an overview:

- It first gets the next token from the lexer, using 'main' lexer function above explained.
- If the token is a PrintKeyword, it expects an opening parenthesis '(', evaluates the expression inside the parenthesis, checks for a closing parenthesis ')', and then prints the result of the evaluated expression.
- If the token is a LetKeyword, it calls the assignment() method to process the assignment of values to variables.
- If the token is an IfKeyword, it calls the Conditional() method to process the conditional instruction.

Some Statements must be processed, for that there is statement():

- Basically does the same as Run(), but with extras
- If the token is an 'Identifier' or 'Number', it does nothing.
- If the token is an 'EOF' (end-of-file), it returns from the method.
- If the token is not recognized, it prints an error message.
- Finally, it checks if there is a semicolon (end-of-line 'EOL' token) and either advances to the next statement or returns the token to the lexer to be analyzed in the next iteration.

Let-In expressions are handled with assignment() method:

- It first gets the next token from the lexer, which should be an identifier (the variable name).
- It then expects an equals '=' operator.
- It evaluates the expression on the right side of the equals operator to get the assigned value.
- It assigns the value to the variable in the variables dictionary.
- It then checks if there is a comma, indicating another assignment statement, and recursively calls assignment() if there is. If there is no comma, it expects an in keyword. Finally, it calls the statement() method to execute the next statement.

Basic conditional statements, processed by Conditional():

- It first gets the next token from the lexer, which should be an opening parenthesis '('.
- It then evaluates the left-hand side of the comparison, expects a comparison operator, and evaluates the right-hand side of the comparison.
- It checks for a closing parenthesis ')'.
- It then performs the comparison operation based on the comparison operator and stores the result.
- If the comparison result is true, it calls the statement() method to execute the next statement.
- If the comparison result is false, it skips to the else part of the if-else statement and executes the statement there.
- If there is no else part, it prints an error message.

Note: RConditional() and RConditional(List¡Token¿ tokens), are simple variations of Conditional(), used on return expressions and functions return expressions respectively.

3.3 Recursive Descent Parser

The jewel in the crown revolves around these functions, they are what allow everything to flow as well as my programming skills allow.

- expression()
  - It first evaluates a term.
  - It then enters a loop where it gets the next token from the lexer and checks if it's an operator.
  - If the operator is @, it evaluates the next term and concatenates it with the left-hand side.
  - If the operator is + or -, it evaluates the next term and performs the binary operation with the left-hand side and the operator.
  - If the token is not an operator, it returns the token to the lexer and returns the left-hand side as the result of the expression.
- term()
  - It first evaluates a power.
  - It then enters a loop where it gets the next token from the lexer and checks if it's a multiplication, division, or modulo operator.
  - If the token is not one of these operators, it returns the token to the lexer and returns the left-hand side as the result of the term.
  - If the token is one of these operators, it evaluates the next power and performs the binary operation with the left-hand side and the operator.
- power()
  - It first evaluates a primary expression.
  - It then enters a loop where it gets the next token from the lexer and checks if it's a power operator (ˆ).
  - If the token is not a power operator, it returns the token to the lexer and returns the left-hand side as the result of the power expression.
  - If the token is a power operator, it evaluates the next primary expression and performs the binary operation with the left-hand side and the operator.

- primary()
  - It first gets the next token from the lexer.
  - It checks if the token is a function call, and if so, it parses the arguments and calls the function.
  - If the token is a number, it returns the number.
  - If the token is a string literal, it returns the string.
  - If the token is an identifier, it returns the value of the variable with that name.
  - If the token is an opening parenthesis (, it evaluates the expression inside the parentheses.
  - If the token is a let keyword, it parses a let-in expression.
  - If the token is an if keyword, it parses an if-else expression.
  - If the token is a minus operator `-`, it negates the next number.
  - If the token is none of the above, it prints an error message.

`Note:` These functions each have an overload specifically designed to handle expressions within functions. Check fnizer.cs

The function that performs Binary Operations on two operands. Simple overview:

- It first checks if the operator token is indeed an operator.
- If the operator is +, it adds the operands if they are both floats or concatenates them if they are both strings.
- If the operator is `-`, it subtracts the operands if they are both floats.
- If the operator is @, it concatenates the operands if they are both strings.
- If the operator is *, it multiplies the operands if they are both floats.
- If the operator is ˆ, it raises the left operand to the power of the right operand if they are both floats.
- If the operator is /, it divides the left operand by the right operand if they are both floats.
- If the operator is %, it calculates the remainder of the division of the left operand by the right operand if they are both floats.
- If the operator is none of the above, it prints an error message.
- If the token is not an operator, it prints an error message.

An extra encapsulation is ConcatenateValues():

- If both operands are strings, it concatenates them.
- If one of the operands is a string, it converts the other operand to a string and then concatenates them.
- If neither operand is a string, it tries to add them if they are both floats.
- If the operands cannot be concatenated or added, it prints an error message.

3.4 Functions

There are 2 types of functions, declared and predefined, declared functions are defined by the user, predefined functions are defined by the 'dev' user, but are not assigned to a variable, they are used to be passed as arguments to other functions, or to be returned by other functions.

3.4.1 Declared Functions

These are the type "function fib(n) => if ($n > 1$) fib($n - 1$) + fib(n-2) else 1", they are defined by the user, and can be assigned to a variable, or passed as an argument to another function. They are defined by the keyword 'function', followed by the name of the function, then the arguments, and finally the body of the function. The body of the function is a series of statements, that can be a return expression, or a basic statement.

These are defined as a "DFunction" class which represents a user-declared function in Hulk. Here's a brief overview of what it does:

- expression: A list of tokens that make up the body of the function.
- value: The name of the function.
- parameters: A list of parameter names for the function.
- type: The type of the function, represented as a TokenType.

The constructor for DFunction takes these four properties as parameters. It also adds an end-of-line token to the end of the function's expression. Small price to pay for salvation.

DFunctions are called on the Recursive Parser section, and are handled by the "EvaFurras()"(I'm so sorry for this name) method:

- It first gets the function's expression tokens and parameter list.
- It checks if the number of arguments matches the number of parameters. If not, it prints an error message.
- It then creates a dictionary mapping each parameter to its corresponding argument.
- It substitutes the arguments into the function's expression using the SubstituteArgs() method, which is presumably defined elsewhere in your code.
- It evaluates the substituted expression using the expression() method, which is also presumably defined elsewhere in your code, and returns the result.

It uses a method called "SubstituteArgs()" it works as follows:

- It creates a new list to hold the substituted tokens.
- It then iterates over each token in the function's expression.
- If the token is an identifier and it's in the argument dictionary, it substitutes the argument value for the identifier.
- If the token is not an identifier or it's not in the argument dictionary, it adds the token to the list as is.
- It returns the list of substituted tokens.

Have in mind that this method is used on the Recursive Parser section, however it uses an overload of the entire Recursive Parser section, that allows it to be used on functions, and not only on the main program.

This 'new' parser, does the same as the main parser, but with some extras, iterations, like processed entires functions expressions and returns the result of the function, or the result of the expression, depending on the context.

3.4.2 Predefined Functions

These are the simple Mathematical Functions explained above, there basically are:

- Name: The name of the function.
- Parameters: The number of parameters the function takes.
- Implementation: The actual implementation of the function, represented as a Func<List<object>, object> delegate.

Data found on fnizer.cs, Functions class. The constructor for Functions takes these three properties as parameters and takes them. All predefined functions are stored in a list, and are called by the primary() method everytime a token coincides with a predefined function name.

This List is:

- It creates a list of Functions objects, each representing a different mathematical function.
- Each Functions object is initialized with a name, the number of parameters it takes, and a lambda function that implements the mathematical operation.
- The lambda function takes a list of arguments, converts them to the appropriate type, and applies the mathematical operation.
- The list includes functions for sine, cosine, tangent, logarithm, natural logarithm, square root, absolute value, power, exponent, floor, ceiling, round, and random number generation. All explained above too.