HULK Interpreter documentation

Eduardo Brito Labrada

October 12, 2023

Abstract

In computer science, an **interpreter** is a computer program that directly executes intructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies for program execution:

1. Parse the source code and perform behavior directly;
2. Translate source code into some efficient intermediate representation or object code and inmediately execute that;
3. Explicitly execute stored precompiled bytecode made by a compiler and matched with the interpreter Virtual Machine.

In this project we will focus on the first strategies of those to create a interpreter for *Havana University Language for Kompilers (HULK)*. First we will define the basic syntax of the language and then we will show how the interpreter works in its entirety.

Contents

Documentation

HULK is a didactic, type-safe, object-oriented and incremental programming language. This is a simplified version of HULK where we will be implementing a subset of this programming language. In particular, this subset consists only of expressions that can be written on one line.

Expressions

HULK is a ultimately an expression-based language. Most of the syntactic constructions in HULK are expressions, including the body of all functions, loops and other block of code. The body of a program in HULK always end with a single global expression (and, if necessary, a final semicolon[1]) that serves as the entrypoint of the program.

For example, the following is a valid program in HULK:

```
42;
```

Obviusly, this program has no side effects. A slightly more complicated program, probably the first one that does something, is this:

```
print(42);
```

In this program, `print` refers to a builtin function that prints the results of any expression in the output stream. We will talk about functions in a later section.

Arithmetic expressions

HULK defines three types of literal values: **numbers**, **strings** and **booleans**. Numbers are 32-bit floating-point and support all basic arithmetic operations with the usual semantic: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (floating-point division), `^` (power), and parenthesized sub-expressions.

The following is a valid HULK program that computes and prints the result of a rather useless arithmetic expression:

```
print(((1 + 2) ^ 3) * 4 / 5);
```

All usual syntactic and procedence rules apply.

Strings

Strings literals in HULK are defined within enclosed double-quotes (`"`), such as in:

```
print("Hello World");
```

A double-quote can be included literally by escaping it (`\"`):

```
print("The message is \"Hello World\"");
```

[1] In this version of HULK all expressions end with a single semicolon

Other escaped characters are `\n` for line endings and `\t` for tabs.

Also, strings can be concatenated with other strings (or the string representation of numbers) using the `@` operator:

```
print("The meaning of life is " @ 42);
```

Builtin math functions and constants

Besides `print`, HULK also provides some common mathematical operations encapsulated as builtin functions with their usual semantic. The list of builtin math functions is the following:

- `sqrt(<value>)` computes the square root if a value.
- `sin<angle>` computes the sine of an angle in radians.
- `cos(<angle>)` computes the cosine of an angle in radians.
- `exp(<value>)` computes the value of `e` raised to a value.
- `log(<base>, <value>)` computes the logarithm of a value in a given base.
- `rand()` returns a random uniform number between 0 and 1 (both inclusive).

Besides these functions, HULK also provides two globals constants: `PI` and `E` wich represent the floating-point value of these mathematical constants.

As expected, functions can be nested in HULK (provided the use of types is consistent, but so far all we care about is functions from numbers to numbers, so we can forget about types until later on). Hence, the following is a valid HULK program:

```
print(sin(2 * PI)^2 + cos(3 * PI / log(2, 4)));
```

More formally, function invocation is also an expression in HULK, so everywhere you expected an expression you can also put a call to builtin function, and you can freely mix arithmetic expressions and mathematical functions, as you would expected in any programming language.

Functions

HULK also lets you define your own functions (of course!). A program in HULK can have an arbitrary number of functions defined before the final global expressions.

A function's body is always an expression, hence all functions have a return value (and type), that is, the return value (and type) of its body.

Inline functions

The easiest way of define a function is the inline form. Here is an example:

```
function tan(x) => sin(x) / cos(x);
```

An inline function is defined by an identifier followed by arguments between parenthesis, then the `=>` symbol, and then a simple expression (not an expression body) as body, ending with a final semicolon.

In HULK, all functions must be defined before the final global expression. All these functions live in a single global namespace, hence is not allowed to repeat function names. Similarly, there are no overloads in HULK (at least in "basic" HULK).

Finally, the body of any function can use other functions, regardless of whether they are defined before or after the corresponding function. Thus, the following is a valid HULK program:

```
function cot(x) => 1 / tan(x);
function tan(x) => sin(x) / cos(x);
```

And of course, inline functions (and any other type of functions) can call themselves recursively.

Variables

Variables in HULK are lexically-scoped, which means that their scope is explicitely defined by the syntax. You use the `let` expression to introduce one or more variables in and evaluate an expression in a new scope where does variables are defined.

The simplest form is introducing a single variable and using a single expression as body.

```
let msg = "Hello World" in print(msg);
```

Here `msg` is a new symbol that is defined only within the expression that goes after `in`.

Multiple variables

The `let` expression admits defining multiple variables at once like this:

```
let x = 42, s = "The meaning of life is " in print(s @ x);
```

This is semantically equivalent to the following long form:

```
let x = 42 in let s = "The meaning of life is " in print(s @ x);
```

As you can notice, `let` associates to the right, so the previous is also equivalent to:

```
let x = 42 in (let s = "The meaning of life is " in (print(s @ x)));
```

Scoping rules

Since the binding is performed left-to-right (or equivalently starting from the outer let), and every variable is effectively bound in a new scope, you can safely use one variable when defining another:

```
let a = 6, b = a * 7 in print(b);
```

Which is equivalent to (and thus valid):

```
let a = 6 in let b = a * 7 in print(b);
```

The `let` return value

As with almost everything in HULK, `let` is an expression, so it has a return value, which is obviusly the return value of its body. This means the following is a valid HULK program:

```
let a = (let b = 6 in b * 7) in print(a);
```

Or more directly:

```
print(let b = 6 in b * 7);
```

Redefining symbols

In HULK every new scope hides the symbols from the parent scope, which means you can redefine a variable name in a inner `let` expression:

And because of the scoping rules, the following is also valid:

```
let a = 7, a = 7 * 6 in print(a);
```

Conditionals

The `if` expression allows evaluating different expressions based on a condition.

```
let a = 42 in if (a % 2 == 0) print("Even") else print("Odd");
```

Since `if` is itself an expression, returning the value of the branch that evaluated true, the previous program can be rewritten as follows:

```
let a = 42 in print(if(a % 2 == 0) "Even" else "Odd");
```

Conditions are just expressions of boolean type. The following are the valid boolean expressions:

- Boolean literals: `true` and `false`
- Arithmetic comparison operators: `<`, `>`, `<=`, `>=`, `==`, `!=`, with their usual semantics.

- Boolean operators: `&` (and), `|` (or), and `!` (not) with their usual semantics.

Errors

In HULK there are three types of errors that must be detected. If an error is found, the interpreter will print one (or more lines) indicating the error in the most informative way possible.

Lexical Error

These types of errors are produced by the presence of invalid tokens. The lexical errors are printed in the following format:

```
! LEXICAL ERROR [{line}:{column}] at ‘{token}’: {message}
```

Here is an example:

```
> let 14a = 5 in print(14a);
! LEXICAL ERROR [1:7] at ‘14a‘: Is not a valid token
! LEXICAL ERROR [1:24] at ‘14a‘: Is not a valid token
```

Syntax Error

These types of errors are produced by miswritten expressions such that unbalanced parenthesis or incomplete expressions. Generally, the syntax errors are printed in the following format:

```
! SYNTAX ERROR [{line}:{column}] at ‘token’: {message}
```

Here are some examples:

```
> let a = 5 in print(a;
! SYNTAX ERROR [1:21] at ‘;‘: Missing closing parenthesis after
   parameters.
> let a = 5 inn print(a);
! SYNTAX ERROR [1:13] at ‘inn‘: Missing ‘in‘ at end of ‘let-in‘
   expression.
! SYNTAX ERROR [1:19] at ‘print‘: Missing ‘;‘.
> let a = in print(a);
! SYNTAX ERROR [1:10] at ‘in‘: Expected some expression but not found.
```

Semantic Error

These types of errors are produced due to incorrect use of types and arguments. Generally, the semantic errors are printed in the following format:

```
> let a = "hello world" in print(a + 5);
! SEMANTIC ERROR: ‘hello world‘ Must be number.
! SEMANTIC ERROR: ‘Expr+Call‘ Unexpected error in ‘let-in‘ expression.
> print(fib("hello world"));
! SEMANTIC ERROR: ‘hello world‘ Must be number.
! SEMANTIC ERROR: ‘Expr+Conditional‘ Unexpected error in ‘let-in‘
   expression.
> print(fib(4, 3))
! SEMANTIC ERROR: ‘fib‘ Incorrect arity for this function.
```

Implementation

In this section each of the phases of the interpreter is explained: scanning, parsing, and evaluating code.

Scanning

The first step in the interpreter is scanning. The scanner takes in raw source code as a series of characters and groups it into a series of **tokens**. These are the meaningful "words" and "puntuaction" that make up the language's grammar.

First of all we need identify all the tokens' types that are used in the language's grammar. Some of this types are:

- **Operators:** `MUL`, `CONCAT`, `MINUS`, `PLUS`, `POWER`, `DIV`, `MOD`.
- **Single-character:** `LEFT_PARENTESIS`, `RIGHT_PARENTESIS`, `COMMA`, `SEMICOLON`.
- **Comparison:** `NOT`, `NOT_EQUAL`, `EQUAL`, `EQUAL_EQUAL`, `GREATER`, `GREATER_EQUAL`, `LESS`, `LESS_EQUAL`.
- **Literals:** `BOOLEAN`, `STRING`, `NUMBER`, `IDENTIFIER`.
- **Keywords:** `AND`, `ELSE`, `FALSE`, `FUNCTION`, `IF`, `LET`, `IN`, `OR`, `TRUE`.
- **Constants:** `EULER`, `PI`.
- **Others:** `IMPLIES`, `EOF`.

Then we iterate the entire line and match each word with its respective token, if the word can be matched with any valid token, we report this as an *lexical error*.

`Scanner.cs`

In this class is implemented the scanning process. We use some auxiliary function:

- `Peek()`: returns the current character in the scanning process.
- `Next()`: returns the next character in the scanning process.

- `Advance()`: returns the current character in the scanning process and advance to the next character.
- `IsAtEnd()`: returns true is the scanning process is at end of the source code.
- `Eat(expected)`: returns true and advance if the current character match with `expected`.
- `IsAlpha(c)`: returns true if `c` is an alphabet character or is equal to _.
- `IsDigit(c)`: returns true if `c` is a digit.
- `IsAlphaNum(c)`: returns true `IsAlpha(c)` or `IsDigit(c)` are true.
- `AddToken(type, literal)`: add token of type `type` and literal `literal` to the list of tokens.

In the scanning process, each character is iterated and an attempt is made to identify the type of token to which it belongs, for this the `ScanToken()` method is used, also `ScanIdentifier()`, `ScanString()`, `ScanNumber()` methods are used to make more easy the process.

title