Beeline Interpreter

SENG 475: Project Proposal

Callum Curtis

V00945190

callumtcurtis@gmail.com

July 21, 2023

1. Introduction

An interpreter accepts human-readable source code and executes it immediately. This is distinct from a compiler, which translates source code into a different representation that can be separately compiled, interpreted, or executed [1]. In this project, an interpreter for a novel programming language, “Beeline”, shall be developed. The deliverable of this project is a Beeline interpreter application that accepts Beeline source code from standard input and executes the program according to the well-defined grammar of the language. The Beeline language and its corresponding interpreter are described below.

2. Project Scope and Limitations

Given the limited time available for the project and the student’s lack of significant background in interpreter development, some industry-standard interpreter features must be omitted. These reductions in scope – and, therefore, limitations of the solution – fall into two categories: omission of language constructs and limiting of execution modes. Refer to section 3.2 Beeline Language Syntax for language constructs that shall be included in the project.

Omitted Language Constructs

- Imports: the interpreter shall operate over the Beeline source code provided through standard input, without the ability to import code from other sources.
- Classes: the Beeline language and interpreter shall not support classes, interfaces, or any other class-related constructs such as polymorphism or inheritance.
- Functions: anonymous and non-anonymous functions shall not be supported by the Beeline language or interpreter.
- Alternative loops (for-each, for, do-while): only a single loop construct, the while-loop, shall be provided by the Beeline language and interpreter. Other loop constructs can be simulated by the user using the provided while-loop construct.

- Standard library utilities (timing, threading, I/O, etc.): there shall not be a standard library supplied with the Beeline language or interpreter; the core functionality of the language shall satisfy the suitability, difficulty, and testability criteria for the project.
- Alternative floating-point literal formats: floating-point literal formats such as “1e-6” shall not be supported. See section 3.2 Beeline Language Syntax for the accepted format.
- Ordering-based comparisons for strings: ordering-based comparisons between strings shall not be supported by the minimum project deliverable and will be considered an optional stretch goal.
- Line continuation capability: the ability to continue a line on the next (e.g., using backslash) shall not be supported by the minimum project deliverable and will be considered an optional stretch goal.

Limited Execution Modes

- Script mode only: the Beeline interpreter shall execute EOF-terminated Beeline programs (script mode), and shall not support interactive interpreter sessions where instructions are executed as soon as they are read by the interpreter (interactive mode) [2].

3. User Interface

The Beeline interpreter application shall be interacted with through the command-line. The remainder of this section describes the interpreter application’s command-line interface and the syntax of Beeline source code.

3.1 Command-Line Interface

Name

beeline – Beeline interpreter

Synopsis

```
beeline [--debug_level=<debug_level>] [--version] [--help]
```

Examples

```
// Interprets Beeline source code in some_input_file.
$ beeline < some_input_file

// Reads Beeline source code from the command-line and executes it only after EOF is read.
$ beeline
```

Description

Interprets Beeline source code provided through standard input. If the version or help options are provided, the corresponding information, described below, shall be output for the user and the process shall exit immediately afterward. Otherwise, the interpreter shall read Beeline source code from standard input until EOF is encountered. After EOF is encountered, the interpreter shall execute the provided program, outputting any print statements (described in 3.2 Beeline Language Syntax) to standard output. For an explanation of why the Beeline interpreter waits until EOF is encountered before evaluating the program, see section 2. Project Scope and Limitations where the lack of support for an interactive mode is discussed. If, during lexing or parsing, a syntax error is encountered, the non-compliant text shall be output for the user alongside a line number and explanation. If, during evaluation, a runtime error is encountered (e.g., division by zero), the erroneous operation will be output for the user alongside a line number and explanation. The Beeline interpreter process shall exit immediately after encountering a syntax or runtime error. The Beeline interpreter application shall only ever write to the standard streams (input, output, error) – never to a file on the user's machine.

Options

- --debug_level
    - Set the debug level to the given value. Acceptable values are 0, 1, 2, 3, 4, or 5, corresponding to trace, debug, info, warning, error, and fatal, respectively. The semantics of these debug levels are consistent with log4j's debug levels [3]. By default, the only

debug information displayed shall be fatal or error information in the specific case that the interpreter exits due to a syntax or runtime error. All debugging information shall be written to standard error (using `std::cerr` or `std::clog`, as appropriate). User output – issued using the print operation to standard output (`std::cout`) – shall not be affected by the debug level.

- --version
    - Output the version information and exit.
- --help
    - Output this help information and exit.

3.2 Beeline Language Syntax

The Beeline programming language is an imperative, general-purpose, interpreted, dynamically typed, high-level scripting language with automatic memory management. This section describes the minimum language features that shall be delivered with the project. The examples given below in `consolas font` denote syntactically-correct Beeline code (although some code, such as variable creation, may be omitted for the sake of clarity).

Data Types

- Boolean
- Double-precision floating point number
- String
- Null

Boolean has two possible values: true and false. The double-precision floating point type shall be the only numerical type. Floating-point literals must be nonnegative and formatted as zero or more digits, optionally followed by a dot alongside one more or more digits (i.e., [0-9]*\.[0-9]+ as a regular expression). String shall represent a sequence of 0 or more characters; a single character shall be representable using a string of length 1 (similarly to Python). Null denotes a lack of value and does not compare equal to anything other than another null (analogous to None in Python).

```
// Acceptable floating-point literals
35.842
.034

// Unacceptable floating-point literals: 1e6
```

Comments

Single-line comments shall be provided using two forward slashes. Multi-line comments shall not be supported. Any text between two forward slashes and the end of the line shall be ignored by the interpreter.

```
// This is a comment. This entire line shall be ignored by the interpreter.
var a = 23  // This is a comment. The "var a = 23" before the slashes shall be interpreted.
```

Arithmetic Operations

Addition, subtraction, multiplication, division, and unary minus operations shall be provided for strictly numerical operands. The results of these operations shall be roughly (due to potential rounding error) consistent with the same operations over two floating-point numbers in C++. Using these operations on any combination of other data types shall result in a runtime error (note that concatenation also uses the "plus" symbol but is a separate operation). The result of an arithmetic operation is a number.

```
// We haven’t introduced how to create/assign variables yet, so refer to the comment beside
// each example for the (current) types of the variables.


a + b  // number + number (addition)

a – b  // number – number (subtraction)

a * b  // number * number (multiplication)

a / b  // number / number (division) (NOTE: division by zero results in runtime error)

-a  // number (unary minus)
```

Concatenation

Concatenation of strings shall be supported. Non-null types can participate in string concatenation when the other operand is of type string; in such cases, the non-string type is implicitly converted to a string before the concatenation is performed. The result of a concatenation operation is a string.

```
// We haven’t introduced how to create/assign variables yet, so refer to the comment beside
// each example for the (current) types of the variables.


x + y  // string + string (concatenation)

x + b  // string + number (implicit conversion of number to string before concatenation)

a + y  // number + string (implicit conversion of number to string before concatenation)

x + t  // string + Boolean (implicit conversion of Boolean to string before concatenation)

t + x  // Boolean + string (implicit conversion of Boolean to string before concatenation)

// string + null is not acceptable
```

Print

The print operation can be used to write a string to standard output (`std::cout`). A runtime error will occur if print’s operand is not of type string. A newline character is not automatically appended to the string.

```
print "The current year is " + 2023  // prints "The current year is 2023" to stdout
```

Comparison Operators

The less-than, less-than-or-equal-to, greater-than, greater-than-or-equal-to, equality, and inequality operations shall be supported. The ordering-based comparisons (less-than, less-than-or-equal-to, greater-than, and greater-than-or-equal-to) shall result in a runtime error if the operands are not strictly numerical. The equality and inequality operations shall be supported between all combinations of types; if the types are different, they shall be considered unequal, otherwise, the values are compared for equivalence. The result of a comparison operation is a Boolean.

```
// Ordering-based comparisons (both operands being numerical)
1 < 2  // less-than
1 <= 1  // less-than-or-equal-to
2 > 1  // greater-than
1 >= 1  // greater-than-or-equal-to

// Ordering-based comparisons involving non-numerical types result in a runtime error

// Equality and inequality (operands of same type)
1 == 2  // false (same type, different values)
1 == 1  // true (same type, same values)
true == false  // false (same type, different values)
true == true  // true (same type, same values)
"123" == "234"  // false (same type, different values)
"123" == "123"  // true (same type, same values)
null == null  // true (null has one possible value and always compares equal to other nulls)

// Equality and inequality operations between different types always evaluate to false
123 == "123"  // false (no implicit conversions performed for equality/inequality checks)
```

Logical Operators

The logical “not”, “and”, and “or” operations shall be supported. The logical “and” and “or” operations shall short-circuit. Only the Boolean type may participate in logical operations. If a non-Boolean type attempts to participate in a logical operation, a runtime error shall occur. Note that values of other types can be converted to a Boolean value using comparison operators. The result of a logical operation is a Boolean.

```
!false  // true
!true  // false

true and false  // false
true and true  // true

true or true  // true
true or false  // true

("123" == "") and (1 < 3)  // false
("123" == "") or (1 < 3)  // true
```

Operator Precedence and Associativity

The operators described above shall have the same precedence and associativity as their corresponding operators in C and C++. Concatenation shall have the same precedence and associativity as addition. Logical “and” and “or” shall have the same precedence and associativity as && and || in C and C++.

Variable Creation

Variables are declared using var. If the variable is not immediately assigned a value, its value defaults to null. Variables can be accessed after declaration using their identifier.

```
var a  // a defaults to null
var b = "hello world!"  // b is a string containing "hello world!"
```

Variable Assignment

Variables can be assigned different values after declaration. Variables are assigned a copy of the operand, meaning that variables cannot reference or alias other variables.

```
var a = 123.456
var b = "foobar"
a = b  // a and b are now both strings containing "foobar"
```

Conditional Statements

Conditional execution shall be supported through if and else statements. The conditions provided to if statements must be of type Boolean. The if statement block shall be enclosed in curly braces.

```
var a = 1
var b = 5
if ((a + 1) < b) {
	print "outer evaluates to true"
	if(a == b) {
		print "inner evaluates to true"
	}
} else {
	print "outer evaluates to false"
}
```

While Loops

Only a single loop variant shall be supported: while-loops. The other loop variants present in other languages (e.g., for-loops) can be simulated by the user using this single loop construct. The while-loop shall execute its body (enclosed in curly braces) repeatedly as long as its condition evaluates to true. The while-loop's condition must evaluate to a Boolean, otherwise a runtime error shall occur.

```
var a = 0
while (a < 5) {
	print (a / 3) + 14
	a = a + 1
}
```

Variable Scope

Variables in Beeline shall have lexical scope. This means that variables declared inside a block (designated using curly braces) cannot be accessed from outside of the block, and that variables in outer blocks can be accessed from within inner blocks.

```
var a = 3
if(a == 3) {
        var b = a  // global variable a is visible from inside any block after a’s declaration
        if(b == a) {
                var c = b + a // a and outer variable b are visible from inner block
        }
        // c is no longer visible
}
// b is no longer visible
// a is still visible
```

4. Non-Student Code

All of the core functionality related to lexing, parsing, and evaluating the Beeline source code shall be written by the student. A handful of utilities shall be employed, however, for tasks that are tangential to the project topic but are necessary for a solid user experience; these utilities are Boost Program Options and Boost Log. Boost Program Options shall be used to format and obtain program options for the beeline application from the command-line. Boost Log shall be used to write debug information and adjust debug levels.

5. High-Level Implementation Description

The implementation shall consist of three key components: a lexer, parser, and evaluator. The lexer shall be responsible for scanning the Beeline source code and converting the source code to tokens that correspond to well-known elements of the language’s context-free grammar. This grammar shall be defined formally using the grammar from “Crafting Interpreters” as a reference [4]. Next, the tokens shall be rearranged into an Abstract Syntax Tree (AST) by the parser. Finally, the evaluator shall traverse this syntax tree using post-order traversal to execute the source code representation; therefore, this project implements a tree-walk interpreter, which is more simple but much slower than other interpreter designs, such as those that compile to bytecode (e.g., Python) [1, 5].

6. Suitability

Design and development of the Beeline language and interpreter represent significant challenges in terms of problem complexity, solution design, and code volume. However, materials like those in the References section represent an abundant supply of background information, advice, and examples online. Additionally, the student is confident and motivated to pursue this topic. The project is also highly testable: the instructor may produce their own Beeline programs and supply those to the interpreter as input, viewing the results through standard output to confirm that the results align with expectations.

7. Work Schedule

Refer to Table 1 for the project work schedule. For more information about implementation components, see section 5. High-Level Implementation Description.

Table 1: Project Work Schedule

Item Name	Effort (1 low, 3 high)	Planned Completion Date
Project Setup (Logging and Program Options)	1	2023-07-26
Formal Grammar Description	3	2023-07-28
Lexer (Tokenization of Source Code)	1	2023-07-29
Parser (Construction of AST)	3	2023-08-02
Evaluator (Traversal and Evaluation of AST)	2	2023-08-06
Demo Video and README.txt	2	2023-08-08
Project Submission	1	2023-08-08

8. Optional Stretch Goals

The optional stretch goals for the project include the language features and execution mode described in section 2. Project Scope and Limitations.

References

[1] “A Map of the Territory · Crafting Interpreters.” https://craftinginterpreters.com/a-map-of-the-territory.html (accessed Jul. 19, 2023).

[2] “2. Using the Python Interpreter,” Python documentation. https://docs.python.org/3/tutorial/interpreter.html (accessed Jul. 20, 2023).

[3] “Log4j Levels Example - Order, Priority, Custom Filters | DigitalOcean.” https://www.digitalocean.com/community/tutorials/log4j-levels-example-order-priority-custom-filters (accessed Jul. 20, 2023).

[4] R. Nystrom, “Crafting Interpreters.” https://craftinginterpreters.com/ (accessed Jul. 19, 2023).

[5] “Chunks of Bytecode · Crafting Interpreters.” https://craftinginterpreters.com/chunks-of-bytecode.html (accessed Jul. 20, 2023).