Implementation of the Haifu Programming Language

In this project I implement the programming language known as Haifu. Haifu is an interpreted language where programs adhere to the form of Haiku poetry. All variables in Haifu are global and can store a sequence of commands or a numeric value. Execution errors are not possible in a Haifu program, although my interpreter warns the user about specific conditions. I coded my interpreter in C++ because I am most familiar with this programming language, and because it allows for fine manipulation of data as well as object orientation. My interpreter provides a command-line interface for the user to run Haifu programs in the form of text files and to edit the **Dataset of Word Data**. To read the original specification of Haifu by its author David Morgan-Mar, go to <http://www.dangermouse.net/esoteric/haifu.html>.

The value of Haifu is that it produces code that not only performs a function, but is also insightful from a natural-language perspective. Many programming languages in the past and present, such as C++ and Java, produce code that certainly performs all manner of useful functions, but this code itself looks quite different from how people naturally express themselves using natural language. As an example of this fact, one can examine the source code of my Haifu interpreter. Haifu code, however, as evidenced by a **Sample Program** below, can perform useful functions while retaining a very natural form of human expression: poetry. This quality ensures that Haifu programs have value not only to a Haifu interpreter or programmer, but also to people unfamiliar with Haifu, or only familiar with natural language, or especially those familiar with poetry. A notable downside, however, is that Haifu programs are inscrutable as code to anyone not familiar with Haifu.

Implementing a Haifu interpreter proved to be a satisfying experience. I elected to write my interpreter fully in C++ instead of using an existing tool so that I could handle the fine intricacies of the Haifu specification, such as handling word data, without using the rule set of an unfamiliar tool. However, I used concepts in my interpreter, such as token generation, that are staples of compilers and interpreters of programming languages. Through working on this interpreter, I have reinforced my knowledge of C++, I have been exposed to one method of writing an interpreter, and I have gained more experiential knowledge of the structure of programming language. The result of my work is the ability to write, execute, and admire artistic Haifu programs.

Interpreter Format

My interpreter determines whether a Haifu program is valid through several steps: **Step 1: Recognition of Haiku Format** of the input text by using the **Dataset of Word Data**, **Step 2: Token Generation** from the data in the file by using the **Dataset of Word Data**, **Step 3: Token Evaluation** of the generated tokens, **Step 4: Program Initialization** from the evaluated tokens, and **Step 5: Program Execution** of the initialized program using the **Dataset of Word Data**. To explain these steps however, I first need to explain several concepts of the interpreter and language.

Program Format

The interpreter treats the program as a list of commands. During execution, the interpreter uses two indices into this list. The first index, called the **Celestial Bureaucrat**, is a program counter that starts at the end of the list and proceeds towards the beginning. The interpreter executes the command at this index and usually proceeds to the next command. The interpreter can increment or decrement this index using certain commands. The second index, called the **Celestial Delegate**, also starts and the end of the list, but it only changes when the interpreter executes certain commands. It functions like a stack pointer or register, providing operands for certain commands. The interpreter ensures that the Delegate is never closer to the beginning of the list than the Bureaucrat. Execution ends when the interpreter executes a specific "program termination" command, like exit() in C++, or when the interpreter sets the position of the Bureaucrat to before the first index of the list.

Dataset of Word Data

The interpreter maintains a mutable dataset of word data with keys of type string. Each entry in the dataset contains the following data: (string) Base Word, (char) Element, (vector of (int)) Syllable Count. The interpreter updates the Dataset of Words by querying the user for the following information for each unknown word it encounters in Step 1: the Base Word of the word, the Element of the word, and the possible Syllable Count values for the word. The Syllable Count can contain multiple values, because the pronunciation of words can vary between people, e.g. aluminium. Thus, when the interpreter is determining whether the syllable count of a line is correct, it must consider all possible syllable counts of all words in that line and determine whether it is possible for that line to have the correct number of syllables.

Dataset of Reserved Words, Dataset of Number Words

The interpreter also maintains two immutable datasets: one for reserved words and another for words indicating numbers. The reserved word dataset has (string) keys and data (char) Code, the latter being unique for each class of reserved word. Each class of reserved words indicates a certain runtime command, and multiple words refer to the same command, e.g. "heaven" and "nirvana" each refer to the "program termination" command. The other dataset has (string) keys and data (int) Number Value, the latter indicating the number value represented by the string. The interpreter initializes both of these at runtime using hard-coded instructions without reading from a file.

Elements

Haifu makes use of a certain property called Element. The Element property of words can be one of the following values: Earth, Metal, Water, Wood, Fire, None (my interpreter uses None to differentiate from Earth when determining the Element of hyphenated words). Each Element has a relationship with each other Element, excepting None, and the relationship between the Element values of operands determines the outcome of several commands. The following cycles represent these relationships:

Create Cycle: Earth, Metal, Water, Wood, Fire

Destroy Cycle: Earth, Water, Fire, Metal, Wood

Fear Cycle: Earth, Wood, Metal, Fire, Water

Love Cycle: Earth, Fire, Wood, Water, Metal

Value Rounding

Whenever the interpreter executes a command that expects an operand to have an integer value, but the operand instead has a fractional value, the interpreter treats it as rounded away from 0, e.g. it treats 1.2 as 2 and -2.2 as -3.

Yin/Yang Value

There are two special numeric properties in Haifu called Yin and Yang. Much like True and False in other languages, their meaning is more important than their exact values. A value has the Yin property if its rounded value is even, and Yang if it is odd. Some commands find the Yin/Yang property of a value and have different effects if it is Yin or Yang.

Interpretation Step 1: Recognition of Haiku Format

The interpreter begins the interpretation process by reading in a Haifu program in the form of a text file of ASCII characters. A valid Haifu program must have stanzas of 3 lines each separated by a single blank line. The first line must contain 5 syllables, the second 7, and the third 5. The interpreter looks up each word it encounters in the **Dataset of Word Data** for its relevant data and keeps track of any words with missing or invalid data, producing an error for each such word. If the interpreter has valid syllable data for each word in a line, it checks whether that line has the appropriate number of syllables, producing an error if it does not. The interpreter also produces an error if a stanza does not have exactly 3 lines. If the interpreter reaches the end of the file and has produced any error in this step, it does not proceed to the next step. At this point, the user should enter any missing word data. If the interpreter does not produce any errors in this step, it passes on the data read from the file in the form of a (vector of (string)) to the next step.

Interpretation Step 2: Token Generation

The interpreter iterates through the text of each line, building a (vector of (Token)).

A Token has the following data: (int) Line Number, (int) Column Number, (string) Name, (char) Type, (int) Value, (char) Element. A Type can be one of the following: Undefined, Reserved Word, Variable, Number, Hyphen, Punctuation, Comment.

The interpreter determines a Token and its Type by recognizing the following kinds of strings:

- Haifu Word
    - A string of characters, each of which being [a-z], [A-Z], or [‘] (apostrophe) OR any other string containing only a single character other than whitespace, end-of-line, hyphen [-], or a numeric character [0-9]
        - If this string is in the **Dataset of Reserved Words**, then it is a Token with a Type of Reserved Word
        - Otherwise, if this string is in the **Dataset of Number Words**, then it is a Token with a Type of Number
        - Otherwise, it is Token with a Type of Variable
    - A string containing only a single character [-] (hyphen)
        - This string is a Token with a Type of Hyphen

- Comment:
    - A string of characters produced by the concatenation of the following:
        - A single character [,] (comma)
        - Every non-comma character before the next comma or end-of-data (spans across lines)
        - If end-of-data is not reached, a single character [,] (comma)
    - My interpreter does not actually concatenate these characters, but suspends Token Generation when it encounters [,] and continues when it encounters the next [,].
    - A Comment is a Token with a type of Comment
- Punctuation:
    - A string containing only a single character that fulfills the following requirements:
        - It is non-alphabetic ([a-z] or [A-Z])
        - It is non-numeric ([0-9])
        - It is non-apostrophe ([‘])
        - It is non-hyphen ([-])
        - It is non-comma ([,])
        - It is non-whitespace
        - It is non-end-of-line
    - Punctuation is a Token with a Type of Punctuation
- Undefined:
    - A string containing only a single character that is numeric ([0-9])
    - Undefined is not a Token, but an error

The data for each Token are as follows:

- Line Number is the index of the line containing its text
- Column Number is the index of the first character of the text in its line
- Name is by default the text which constitutes the token
    - If the Type of the Token is Comment, Name is “,”
- Value is by default 0
    - If the Type of the Token is Reserved Word, Value is the Code corresponding to the Name entry in the **Dataset of Reserved Words**
    - If the Type of the Token is Number, Value is the Number Value corresponding to the Name entry in the **Dataset of Number Words**
- Element is by default None
    - If the Type of the Token is Number, Element is Earth
    - If the Type of the Token is Variable or Reserved Word, Element is in the **Dataset of Word Data** or **Dataset of Reserved Words**.

Interpretation Step 3: Token Evaluation

After the interpreter iterates through the text, it then evaluates the Tokens in the following manner:

- Any Token with type Undefined produces an error.
- Hyphen at the beginning or end of the program produces an error.
- Hyphen next to Punctuation, Comment, or another Hyphen produces an error.
- Any sequence of Variable, Number, or Reserved Word tokens delimited by Hyphen: replace by a new Token with the following data:
    - Line Number is the Line Number of the first Token
    - Column Number is the Column Number of the first Token
    - Name is the concatenation of the Name of each Token (including hyphens)
    - Type depends on the Type of all Tokens that do not have the Type of Hyphen
        - If each Type is Number, then Type is Number
        - Otherwise, Type is Variable
    - Value depends on the Type determined above
        - If Type is Number, then Value is the combination of the Value of each Token that does not have the Type of Hyphen (the combination algorithm is shown below)
        - Otherwise, Value is 0
    - Element depends on the Element of all Tokens that do not have the Type of Hyphen
        - If each Token has the same Element, then Element is that Element
        - Otherwise, Element is Earth

The Value of combined Number tokens is determined by the following algorithm. I demonstrate this algorithm using the number 123,456, which in Haifu code would be "one-hundred-twenty-three-thousand-four-hundred-fifty-six". I indicate a combined value with *. My interpreter generates the following sequence of Number tokens from this code: {1, 100, 20, 3, 1000, 4, 100, 50, 6}. The algorithm is as follows (the magnitude of a value is floor($\log_{10}$(value))):

1. Sum any value with magnitude less than 2 that is divisible by 10 with the following value if the following value has a magnitude of 0.

   (Result: {1, 100, 23*, 1000, 4, 100, 56*})

   a. A value with magnitude less than 2 that is not divisible by 10 or that is followed by a value with the same magnitude is an error.
2. Multiply any value with magnitude less than 2 with the following value if the following value has a magnitude of 2.

   (Result: {100*, 23, 1000, 400*, 56})

   a. A value with magnitude less than 2 followed by a value with magnitude less than 2 is an error.
3. Sum any value with magnitude 2 with the following value if the following value has a magnitude less than 2.

   (Result: {123*, 1000, 456*})

   a. A value with magnitude 2 followed by a value with magnitude 2 is an error.
4. Multiply together all sequences of values with strictly ascending magnitude.

   (Result: {123000*, 456})

   a. Two adjacent values with the same magnitude is an error.

5. Sum together all sequences of values with strictly descending magnitude.

   (Result: {123456*})

   a. Two adjacent values with the same magnitude is an error.

6. The first of the remaining values is the Value of the token.

   (Result: the single value 123,456)

   a. The presence of more than one value is an error.

Once the interpreter generates and evaluates all Tokens, it removes all Tokens with the Type of Comment. If the interpreter produces no errors in this step, it passes the existing Token data in the form of a (vector of (Token)) to Step 4. If an error was produced, it instead passes an empty vector.

Interpretation Step 4: Program Initialization

The interpreter iterates through the (vector of (Token)) and for each Token appends a Rung to an initially empty (vector of (Rung)) that represents the program.

A program command is in the form a Rung which has the following data: (int) Line Number, (int) Column Number, (string) Name, (char) Type, (double) Value, (char) Element. Type can be one of the following: Undefined, Command, Variable, Literal, Punctuation.

The data for each Rung as the interpreter iterates through the (vector of (Token)) are determined as follows:

- Line Number is the Line Number of the Token
- Column Number is the Column Number of the Token
- Name is the Name of the Token
- Type depends on the Type of the Token
    - For a Reserved Word token, Type is Command
    - For a Variable token, Type is Variable
    - For a Number token, Type is Literal
    - For a Punctuation token, Type is Punctuation
- Value depends on the Type of the token
    - For a Reserved Word or Number Token, Value is the (double)Value of the token
    - For a Variable or Punctuation token, Value is 0.0
- Element depends on the Type of the token
    - For a Reserved Word, Number, or Punctuation token, Element is Earth
    - For a Variable token, Element is the Element of the token

Once the interpreter finishes iterating through the Tokens, it reverses the program, clears the **Dataset of Variables**, and sets the Bureaucrat and Delegate indices to 0 (so end-to-beginning progression of these indices becomes equivalent to iterating through the program starting from index 0).

Interpretation Step 5: Program Execution

The interpreter then executes the program by iterating through the (vector of (Rung)) that represents the program:

1. While Bureaucrat has not progressed past the upper bound of the program list
    a. Indicate that Bureaucrat should progress
    b. Execute the Rung at Bureaucrat
    c. If this execution indicates program termination
        i. Terminate program
    d. If the Bureaucrat should still progress
        i. Increment Bureaucrat by 1

Dataset of Variables

The interpreter maintains a dataset of variables with (string) keys and data (Variable). A Variable has the following data: (bool) isCommand, (double) Value, (char) Element, (vector of (Rung)) Commands. This dataset is cleared for each Program Execution.

Corresponding Variable

When the interpreter determines the **Corresponding Variable** of a Rung with a Type of Variable, it looks up the entry in the **Dataset of Word Data** corresponding to the Name field of the Rung. If the Base Word of this entry is the default Base Word, then the **Corresponding Variable** is the entry in the **Dataset of Variables** corresponding to the Name of the Rung; otherwise the **Corresponding Variable** is the entry in the **Dataset of Variables** corresponding to this Base Word.

Same Name

When the interpreter needs to check whether two Rungs have the **Same Name**, it checks if they both have the type of Variable and if the **Corresponding Variable** of each of them is the same OR if they both have the Type of Reserved Word and if their Value fields have the same value OR if they both have the Type of Literal and if their Value fields have the same value. If so, they have the **Same Name**; otherwise, they do not have the **Same Name**.

True Value

The **True Value** of a Rung with the Type of Literal is its Value. The **True Value** of a Rung with the Type of Variable is the Value field of its **Corresponding Variable** if the isCommand field of this Variable is False. In any other case, the Rung has no **True Value**.

True Element

The **True Element** of a Rung with a Type that is not Variable is its Element. The **True Element** of a Rung with the Type of Variable is the Element field of the entry in **Dataset of Variables** corresponding to its Name.

Variable Execution

When the interpreter executes a Rung with the Type of Variable, it accesses its **Corresponding Variable**. If the isCommand field of this Variable is True, the interpreter iterates through each Rung in its Commands field and executes them; if isCommand is False, then nothing happens. If any Rung execution indicates program termination, then Program Execution terminates.

Punctuation Execution

When the interpreter executes a Rung with the Type of Punctuation, it remembers the Rung above the Bureaucrat (at index + 1 after reversal). It then iterates through each Rung above the Bureaucrat until it encounters a Rung with the Type of Punctuation or a Rung with the **Same Name** as the remembered Rung. If the remembered Rung has the Type of Variable, it then creates a (vector of (Rung)) containing each Rung between these two and assigns it to the Commands field of the entry in the **Dataset of Variables** corresponding to the Name of the remembered Rung. It also assigns True to the isCommand field of this same entry. The interpreter then sets the Bureaucrat index to the Rung last encountered. If this Variable has not before been assigned a **True Value** or sequence of commands, the **True Element** of the Rung with Type of Variable is set to the Element of the Rung (initializes the Element field in the **Dataset of Variables** entry).

Command Execution

When the interpreter executes a Rung with the Type of Command, it interprets its Value field as one of the **Command Codes** and executes a function corresponding to this command.

Command Codes:

(“above” indicates increasing index and “below” indicates decreasing index)

Heaven

Program Execution terminates.

Promote

The interpreter checks if the Rung below Bureaucrat has a **True Value**. If it does and its **True Value** is not 0, the interpreter increments Bureaucrat by this value and indicates that Bureaucrat should not progress. If the index of Bureaucrat is now beyond the upper bound of the program, the interpreter sets it to just after the last Rung in the program, and this command indicates program termination. If the index of Bureaucrat is now beyond the lower bound of the program, the interpreter assigns 0 to Bureaucrat. If the index of Bureaucrat is now below that of Delegate, the interpreter sets Delegate to Bureaucrat.

Demote

Similar to **Promote**, but decrements Bureaucrat instead of incrementing it.

Blossom

Similar to Promote, but increments Bureaucrat by the value if its **Yin/Yang Value** is Yin, and decrements Bureaucrat if it is Yang.

Rise

The interpreter checks if the Rung below Bureaucrat has a **True Value**. If it does and this value is not 0, the interpreter increments Delegate by this value. If it does not have a **True Value**, the interpreter increments Delegate by 1. If the index of Bureaucrat is now below that of Delegate, the interpreter sets Delegate to Bureaucrat. If the index of Delegate is now beyond the lower bound of the program, the interpreter assigns 0 to Delegate.

Fall

Similar to **Rise**, but decrements Delegate instead of incrementing it.

Listen

The interpreter checks if there are any characters in the input stream and whether they represent a decimal value. If there are characters, the interpreter inserts at the start of the program vector (index 0) a Rung with a Type of Literal, Value of the decimal or character value indicated by the input characters, Name of the input text, and Element of Earth. It then removes this value from the input stream and increments the indices of Bureaucrat and Delegate by 1; otherwise, the interpreter checks if Bureaucrat is at the end of the program vector. If it is not, and if there is a Rung above Bureaucrat, the interpreter removes this Rung, inserts it at the start of the program vector, and increments the indices of Bureaucrat and Delegate by 1.

Speak

The interpreter checks if the Rung at the Delegate has a **True Value**. If it does and this value corresponds to a valid character (an ASCII character in my interpreter), the interpreter outputs this character.

Count

The interpreter checks if the Rung at the Delegate has a **True Value**. If it does, the interpreter outputs a string of numerals representing its value.

Create

The interpreter checks if the Rung at the Delegate is a Variable that has a **True Value**. If it is, it changes its **True Element** to the next in the **Create Cycle**.

Destroy

The interpreter checks if the Rung at the Delegate is a Variable that has a **True Value**. If it is, it changes its **True Element** to the next in the **Destroy Cycle**.

Fear

The interpreter checks if the Rung at the Delegate is a Variable that has a **True Value**. If it is, it changes its **True Element** to the next in the **Fear Cycle**.

Love

The interpreter checks if the Rung at the Delegate is a Variable that has a **True Value**. If it is, it changes its **True Element** to the next in the **Love Cycle**.

Become

The interpreter checks if the Rung at the Delegate has a **True Value**. If it does, the interpreter checks its **True Value**. If it is 0, the interpreter changes the Type of this Rung to Command and its Value to Heaven; if it is an integer, the interpreter changes its **True Value** by 1.0 so it is farther away from 0; otherwise, the interpreter rounds its **True Value** away from 0. The interpreter then checks if the Rung at the Delegate is a Variable. If it is, the interpreter changes its **True Element** to the next in the Create Cycle.

Like

The interpreter checks if the Rung below the Bureaucrat is a Variable. If it is, the interpreter finds the first Rung at or below the Delegate that has a **True Value**. If it finds one, it assigns the **True Value** of this found Rung to the **True Value** of the Rung with Type of Variable and lowers the Delegate to the position of the found Rung; otherwise, it assigns 0 to the **True Value** of the Rung with Type of Variable. If this Variable has not before been assigned a **True Value** or sequence of commands, the **True Element** of the Rung with Type of Variable is set to the Element of the Rung (initializes the Element field in the **Dataset of Variables** entry).

Tomorrow

Nothing happens.

(This command is included in the specification but is not defined there. I include it here to reflect the specification.)

Negative

The interpreter checks if the Rung at the Delegate has a **True Value**. If it does and the Rung is a Variable or Literal, the interpreter changes the **True Value** of this Rung to the negation of this value.

Operate

The interpreter checks if the Rung at the Delegate has a **True Value**. If it does, the interpreter checks if the Rung above the Delegate has a **True Value**. If it also does, the interpreter performs one of the following operations under the given conditions, with B representing the Rung at the Delegate, and A representing the Rung above the Delegate. The conditions depend on the Elements of A and B. Afterwards, the interpreter changes the **True Value** of B to the returned result, if possible.

(A is right after B in the **Create Cycle**): return the sum of A and B (A + B).

(A is right after B in the **Destroy Cycle**): return the difference of A and B (A - B).

(A is right after B in the **Fear Cycle**): return the quotient of A and B (A / B).

(A is right after B in the **Love Cycle**): return the product of A and B (A * B).

(A and B have the same Element): check the Yin/Yang values of A and B. If both are Yang, return Yang; otherwise, return Yin.

Sample Program: echo_number.txt

This Haifu program reads the first value from the input stream and outputs it in decimal format. If there is no input, it outputs “0”.

```
Heaven counts, it does,
A man falls once, into hell,
None listen, 'tis true
```

As you can see, in addition to performing this task, the program itself offers a religious musing. Here is the trace of this program that my interpreter produces when given the input “3.14159”:

```
------------------------------------------------------------
Execution 0:
------------------------------------------------------------
    7: (0, 0) "heaven" COMMAND HEAVEN earth
    6: (0, 7) "counts" COMMAND COUNT earth
    5: (1, 0) "a" LITERAL 1 earth
    4: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
    3: (1, 6) "falls" COMMAND FALL earth
    2: (1, 12) "once" LITERAL 1 earth
    1: (2, 0) "none" LITERAL 0 earth
D>B>0: (2, 5) "listen" COMMAND LISTEN earth

------------------------------------------------------------
Variables at execution 0:
------------------------------------------------------------

------------------------------------------------------------
Execution 1:
------------------------------------------------------------
    8: (0, 0) "heaven" COMMAND HEAVEN earth
    7: (0, 7) "counts" COMMAND COUNT earth
    6: (1, 0) "a" LITERAL 1 earth
    5: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
    4: (1, 6) "falls" COMMAND FALL earth
    3: (1, 12) "once" LITERAL 1 earth
  B>2: (2, 0) "none" LITERAL 0 earth
D>  1: (2, 5) "listen" COMMAND LISTEN earth
    0: (INPUT, 0) "3.14159" LITERAL 3.14159 earth

------------------------------------------------------------
Variables at execution 1:
------------------------------------------------------------

------------------------------------------------------------
Execution 2:
------------------------------------------------------------
    8: (0, 0) "heaven" COMMAND HEAVEN earth
    7: (0, 7) "counts" COMMAND COUNT earth
    6: (1, 0) "a" LITERAL 1 earth
    5: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
    4: (1, 6) "falls" COMMAND FALL earth
  B>3: (1, 12) "once" LITERAL 1 earth
    2: (2, 0) "none" LITERAL 0 earth
D>  1: (2, 5) "listen" COMMAND LISTEN earth
    0: (INPUT, 0) "3.14159" LITERAL 3.14159 earth
```

```
------------------------------------------------------------
Variables at execution 2:
------------------------------------------------------------

------------------------------------------------------------
Execution 3:
------------------------------------------------------------
    8: (0, 0) "heaven" COMMAND HEAVEN earth
    7: (0, 7) "counts" COMMAND COUNT earth
    6: (1, 0) "a" LITERAL 1 earth
    5: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
  B>4: (1, 6) "falls" COMMAND FALL earth
    3: (1, 12) "once" LITERAL 1 earth
    2: (2, 0) "none" LITERAL 0 earth
D>  1: (2, 5) "listen" COMMAND LISTEN earth
    0: (INPUT, 0) "3.14159" LITERAL 3.14159 earth

------------------------------------------------------------
Variables at execution 3:
------------------------------------------------------------

------------------------------------------------------------
Execution 4:
------------------------------------------------------------
    8: (0, 0) "heaven" COMMAND HEAVEN earth
    7: (0, 7) "counts" COMMAND COUNT earth
    6: (1, 0) "a" LITERAL 1 earth
  B>5: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
    4: (1, 6) "falls" COMMAND FALL earth
    3: (1, 12) "once" LITERAL 1 earth
    2: (2, 0) "none" LITERAL 0 earth
    1: (2, 5) "listen" COMMAND LISTEN earth
D>  0: (INPUT, 0) "3.14159" LITERAL 3.14159 earth

------------------------------------------------------------
Variables at execution 4:
------------------------------------------------------------
```

```
-------------------------------------------------------------
Execution 5:
-------------------------------------------------------------
    8: (0, 0) "heaven" COMMAND HEAVEN earth
    7: (0, 7) "counts" COMMAND COUNT earth
  B>6: (1, 0) "a" LITERAL 1 earth
    5: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
    4: (1, 6) "falls" COMMAND FALL earth
    3: (1, 12) "once" LITERAL 1 earth
    2: (2, 0) "none" LITERAL 0 earth
    1: (2, 5) "listen" COMMAND LISTEN earth
D>  0: (INPUT, 0) "3.14159" LITERAL 3.14159 earth

-------------------------------------------------------------
Variables at execution 5:
-------------------------------------------------------------

-------------------------------------------------------------
Execution 6:
-------------------------------------------------------------
    8: (0, 0) "heaven" COMMAND HEAVEN earth
  B>7: (0, 7) "counts" COMMAND COUNT earth
    6: (1, 0) "a" LITERAL 1 earth
    5: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
    4: (1, 6) "falls" COMMAND FALL earth
    3: (1, 12) "once" LITERAL 1 earth
    2: (2, 0) "none" LITERAL 0 earth
    1: (2, 5) "listen" COMMAND LISTEN earth
D>  0: (INPUT, 0) "3.14159" LITERAL 3.14159 earth

-------------------------------------------------------------
Variables at execution 6:
-------------------------------------------------------------

Output: "3.14159"
```

```
-------------------------------------------------------------
Execution 7:
-------------------------------------------------------------
  B>8: (0, 0) "heaven" COMMAND HEAVEN earth
    7: (0, 7) "counts" COMMAND COUNT earth
    6: (1, 0) "a" LITERAL 1 earth
    5: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
    4: (1, 6) "falls" COMMAND FALL earth
    3: (1, 12) "once" LITERAL 1 earth
    2: (2, 0) "none" LITERAL 0 earth
    1: (2, 5) "listen" COMMAND LISTEN earth
D>  0: (INPUT, 0) "3.14159" LITERAL 3.14159 earth

-------------------------------------------------------------
Variables at execution 7:
-------------------------------------------------------------

-------------------------------------------------------------
Termination:
-------------------------------------------------------------
  B>8: (0, 0) "heaven" COMMAND HEAVEN earth
    7: (0, 7) "counts" COMMAND COUNT earth
    6: (1, 0) "a" LITERAL 1 earth
    5: (1, 2) "man" VARIABLE UNINITIALIZED_VARIABLE earth
    4: (1, 6) "falls" COMMAND FALL earth
    3: (1, 12) "once" LITERAL 1 earth
    2: (2, 0) "none" LITERAL 0 earth
    1: (2, 5) "listen" COMMAND LISTEN earth
D>  0: (INPUT, 0) "3.14159" LITERAL 3.14159 earth

-------------------------------------------------------------
Variables at termination:
-------------------------------------------------------------
```

Sample Program: echo_letter.txt

This Haifu program reads the first value from the input stream and outputs it in ASCII format. If there is no value, it outputs the NULL character.

```
Heaven speaks, it does,
A man falls once, into hell,
None listen, 'tis true
```

This program additionally offers a religious musing similar to the previous program.

Sample Program: echo_letters10.txt

This Haifu program reads the first 10 values from the input stream and outputs them in ASCII format. If there are fewer than 10 values, it outputs the NULL character in the place of any missing values.

```
Zero, the thing gone,
Repeating, what is nothing,
One, the thing is here,

None repeat a thing
Nothing repeating a thing
None repeat a thing

None repeat a thing
Nothing repeating a thing
None repeat a thing

None repeat a thing
Nothing repeating a thing
None repeat a thing

Who repeats drawing
He who falls, then stands again,
Listen and repeat.
```

Additionally, one can add or remove copies of the second stanza to respectively increase or decrease the number of values read and output by 3 values for each copy added or removed.

Closing Remarks

Implementing an interpreter for the Haifu programming language has been an enjoyable endeavor, as has been programming in Haifu. Haifu certainly deviates from commonly used programming languages in many aspects, such as requiring a strict form and syllable count, but these aspects make it endearing. Never before have I been able to look at a program and admire it for its poetic value as well as its functional value. Haifu is definitely not a practical language, but it produces programs that can be enjoyable to read purely as poetry.