Assignment 1: The Feeny Teaching Language

Patrick S. Li

August 12, 2015

1 Introduction

The Feeny programming language is an object oriented, imperative, dynamically typed language designed to have similar core functionality as many currently popular scripting languages (such as Python, Javascript, Ruby, etc.) while remaining lean enough to be implemented by a single student in a semester long course.

It was consciously designed to allow a naive and slow interpreter to be implemented concisely, whilst requiring sophisticated dynamic compilation techniques to obtain good performance. Through the course, students will develop multiple implementations of Feeny, each with increasing sophistication and performance, starting with a simple abstract syntax tree (AST) interpreter, and ending with a just-in-time (JIT) compiler.

This initial assignment will help students familiarize themselves with the syntax and semantics of the Feeny language.

2 Running the Sample Programs

The included zip file includes a minimal Feeny interpreter and some sample test programs for getting started. To run the provided Hello World test, type the following into the terminal:

```
./feeny -e tests/hello.feeny
```

That should print

```
Hello World
```

to the screen.

3 Lexical Structure of Feeny

The current implementation of Feeny borrows the lexer of the Stanza programming language, and has the same features and restrictions. Please pay attention to the following properties:

3.1 Comments

Comments in Feeny are indicated by a semicolon and proceed to the end of the line.

```
var c = 0         ;Initialize c to zero
while c < 9 :     ;Proceed when c is less then nine
   begin-check()  ;Check board.
```

3.2 Indentation Structuring

Feeny uses indentation to indicate structuring through the following technique: a *line ending* colon automatically surrounds the following indented block with parenthesis.

Thus the following

```
while i < 10 :
   do-this()
   do-that()
```

is *equivalent* to

```
while i < 10 : (do-this() do-that())
```

3.3 Commas act as Whitespace

Commas in Feeny are treated identically to spaces, and are used solely for readability.

Thus the following

```
f(a, b, c)
```

is equivalent to

```
f(a b c)
```

3.4 Opening Parenthesis

Identifiers *immediately* followed by opening parenthesis and braces are differentiated from identifiers followed by whitespace. This most commonly occurs in the syntax for function calls and array accesses.

The following code:

```
f(a(1 + 2))
```

is the result of calling the function `f` with a single argument: the result of calling the function `a` with `1 + 3`.

In contrast, the following code:

```
f(a (1 + 2))
```

is the result of calling the function `f` with two arguments. The first argument is `a`, and the second argument is `(1 + 2)`.

4 Feeny Language Constructs

4.1 Expressions

Integer Literal: A special form for indicating a 32 bit integer.

```
42
```

Variable Reference: A special form used for referring to a local or global variable.

```
x
```

Printing: A special form for printing out the values of a list of expressions according to a supplied format string.

```
printf("My age is ~.\n", 8)
```

Arrays: A special form for creating arrays of a given length with some initial value.

```
array(10, 0)
```

Null: A special object that refers to the empty object with no slots.

```
null
```

Objects: A special form used for creating objects given a parent object and a list of slots. See section 4.2 for a description of possible slots.

```
object(p) :
   var x = 10
   var y = 10
   method f () :
      this.x + this.y
```

The parent object may be omitted, in which case the default parent will be `null`.

```
object :
   var x = 10
   var y = 10
   method f () :
      this.x + this.y
```

Method Calls: A special form used for calling a named method in a given object with a supplied list of arguments.

```
o.f(10)
```

Slot Lookup: A special form used for looking up a variable slot in a given object.

```
o.field
```

Slot Assignment: A special form used for assigning a new value to a variable slot in a given object.

```
o.field = 42
```

Function Call: A special form is used for calling a named function with a supplied list of arguments.

```
f(10)
```

Variable Assignment: A special form used for assigning to a local or global variable.

```
x = 42
```

If Expression: A special form used for testing against a predicate and then conditionally evaluating one of two expressions.

```
if x :
   printf("A")
else :
   printf("B")
```

The `else` branch may be optionally omitted, in which case the default behaviour is to simply return `null`.

```
if x :
   printf("A")
```

is equivalent to

```
if x :
   printf("A")
else :
   null
```

While Expression: A special form used for repeatedly executing an expression so long as a given predicate is not null.

```
while p :
   x = x + y
```

4.2 Object Slots

Slot statements are used within `object` bodies for implementing object state and behaviour.

Variable Slot: Variable slots are defined by a given name and an initializing expression.

```
var x = 42
```

Method Slot: Method slots are defined by the method name, the argument list, and a statement for the method body.

```
method f (x, y, z) :
   x + y
```

4.3 Local Statements

Local statements are used to represent function and method bodies, bodies for the consequent and alternate branches in if expressions, and loop bodies for while expressions.

Local Variable: Local variables are defined given a name and an initializing expression.

```
var x = 42
```

Sequence of Statements: A sequence of statements can be grouped together by wrapping them in parenthesis.

```
(s1 s2 s3 ...)
```

Local Expression: An expression can be used as a statement. Note that the last statement in a method or function body must be an expression, and indicates the return value of the method or function.

```
f(x)
```

4.4 Top Level Statements

Global Variable: Global variables are defined given a name and an initializing expression.

```
var x = 42
```

Function: A function is defined by a given name, a list of arguments, and a statement for the function body.

```
defn f (x, y, z) :
   x + y
```

Sequence of Statements: A sequence of statements can be grouped together by wrapping them in parenthesis.

```
(s1 s2 s3 ...)
```

Top Level Expression: An expression can be used as a top level statement.

```
f(x)
```

5 Syntactic Shorthands in Feeny

The listing of constructs in the previous section is exhaustive and details every construct in the Feeny language. However, for convenience and readability, a number of syntactic shorthands exist for common operators that expand into one of the core constructs.

```
      x + y  expands into  x.add(y)
      x - y  expands into  x.sub(y)
      x * y  expands into  x.mul(y)
      x / y  expands into  x.div(y)
      x % y  expands into  x.mod(y)
      x < y  expands into  x.lt(y)
      x > y  expands into  x.gt(y)
     x <= y  expands into  x.le(y)
     x >= y  expands into  x.ge(y)
     x == y  expands into  x.eq(y)
     x[y,z]  expands into  x.get(y,z)
x[y,z] = w  expands into  x.set(y,z,w)
```

6 Operational Semantics of Feeny

The following rules informally describe the operational semantics of Feeny.

6.1 Interpreter Structures

Null: Null is a distinguished object that represents an environment with no slots.

Environment: An Environment consists of a list of name-to-EnvValue pairs along with a parent which may be either Null or another Environment. Environments are used to represent function activation records as well as objects. An EnvValue can be either a VarValue or a CodeValue.

- **VarValue**: A VarValue represents the storage location for a variable and must support the following two operations: retrieving the current value of the variable, and storing a new value for the variable.
- **CodeValue**: A CodeValue represents a function or a method. It consists of a list of argument names and the statement corresponding to the body of the function or method.

An Environment supports the following two operations:

1. Adding a new binding. This is done by adding a new name-to-EnvValue pair to the list of bindings in the environment.
2. Retrieving an existing binding by name. This is done by searching the list of name-to-EnvValue pairs for one that matches the requested name. If a pair is not found then the lookup is continued in the parent environment.

Integers: An integer is assumed to be 32-bit number. An integer supports the following arithmetic operations:

1. Add
2. Subtract
3. Multiply
4. Divide
5. Modulo
6. Equality
7. Less Than
8. Less Than or Equal
9. Greater Than
10. Greater Than or Equal

Arrays: An array represents an ordered sequence of storage locations of a fixed size. An array is created given a size and an initial value. It supports the following primitive operations:

1. Retrieving the value at a given index.
2. Storing a value at a given index.
3. Querying the length of the array.

6.2 Evaluation Rules

For the following evaluation rules, an implicit global environment called *genv* is assumed to be present for all rules which represents the top-level Environ-

ment. Rules are written assuming a list of assumptions is satisfied. If the assumptions are not satisfied, then the evaluation is considered stuck. Implementations are required to print an appropriate error message, and quit gracefully in these circumstances.

Evaluating Expressions

Details the rules for evaluating an expression, *e*, inside an environment *env*. In the description for each rule, when a particular expression is said to be evaluated, it is assumed to be evaluated in the environment *env* unless otherwise stated.

Integers: Integer literals evaluate to the value that they represent.

Variable Reference: Assuming that a binding is retrievable in env with the name of the variable, and that the binding is a VarValue, a reference evaluates to the stored value in the VarValue.

Printing: The arguments are first evaluated in order. Then the format string is printed to the screen where occurrences of tildes (~) is replaced with an argument. The print expression evaluates to Null.

Array Creation: The given length and initial value are first evaluated. Assuming that the length is a non-negative integer, then the expression evaluates to an array with the resultant length with every location set to the initial value.

Null: Evaluates to Null.

Objects: The parent object, and all initializing expressions corresponding to variable slots are evaluated. Assuming that the parent is either Null or an Environment, the expression evaluates to a new Environment with the given parent with bindings to VarValues for all variable slots, and bindings to CodeValues for all method slots.

Method Calls: The receiver, and the arguments are evaluated. Depending on the type of the receiver different rules apply.

- **Method Calls for Integers**: For methods add, sub, mul, div, and mod, assuming that there is a single argument, the expression evaluates to the arithmetic result of the corresponding operation.

For methods eq, lt, le, gt, ge, assuming there is a single argument, the expression evaluates to 0 if the corresponding primitive relation holds, or Null if it does not.

- **Method Calls for Arrays**: For method length, assuming no arguments, the expression evaluates to the length of the array.

  For method get, assuming a single integer argument index that is within bounds of the array, the expression evaluates to the result of retrieving the value at the given index.

  For method set, assuming two arguments, the first of which is an integer index within bounds of the array, the expression stores the second argument into the array at the given index, and evaluates to Null.

- **Method Calls for Environments**: Assume that a binding is retrievable in the receiver with the method name, the binding is a CodeValue, and the number of arguments given matches the number of arguments expected by the CodeValue. The method call evaluates to the result of evaluating the body under a new environment:

  1. With parent equal to the global environment, genv.
  2. With a binding from "this" to a VarValue containing the receiver object.
  3. With a binding from each argument listed in the CodeValue to the respective argument in the method call.

Slot Lookup: The receiver object is evaluated. Then assuming that the receiver is an Environment, that a binding is retrievable in the receiver with the given slot name, and that the binding is a VarValue, the expression evaluates to the value stored in the VarValue.

Slot Assignment: The receiver object, and the value is evaluated. Then assuming that the receiver is an Environment, that a binding is retrievable in the receiver with the given slot name, and that the binding is a VarValue, the expression stores the value in the VarValue, and evaluates to that value.

Function Call: The arguments are evaluated. Assume that a binding is retrievable in genv with the name of the function, that the binding is a CodeValue, and that the number of arguments given matches the number of

arguments expected by the CodeValue. The function call evaluates to the result of evaluating the body under a new environment:

1. With parent equal to the global environment, genv.
2. With a binding from each argument listed in the CodeValue to the respective argument in the function call.

Variable Assignment: The value to assign is evaluated. Assume that a binding is retrievable in env with the name of the variable and that the binding is a VarValue. The expression stores the value into the VarValue and evaluates to that value.

If Expression: If the predicate evaluates to Null, then the if expression evaluates to the result of evaluating the alternate statement under a new Environment with parent equal to env. Otherwise, the expression evaluates to the result of evaluating the consequent statement under a new Environment with parent equal to env.

While Expression: If the predicate evaluates to Null, then the loop is finished and the expression evaluates to Null. Otherwise, the body is evaluated in a new Environment with parent equal to env, and then the while expression is re-evaluated under env for the next iteration.

Evaluating Local Statements

Details the rules for evaluating a local statement, *s*, given the local environment, *env*.

Local Variable: Assume that a binding with the given name does not already exist in the *top frame* of env. Then the initializing expression is evaluated and a new binding from the variable name to a VarValue containing the initial value is added to the Environment.

Local Expression: The expression is evaluated under env. The statement evaluates to the result of evaluating the expression.

Sequence of Statements: The statements are evaluated sequentially in order. Note that the last statement in a function body, method body, if body, or while body, must be an expression.

Evaluating Top Level Statements

Details the rules for evaluating a top level statement, *s*.

Global Variable: Assume that a binding with the given name does not already exist in the genv. Then the initializing expression is evaluated and a new binding from the variable name to a VarValue containing the initial value is added to genv.

Function: Assume that a binding with the given name does not already exist in genv. Then a new binding from the function name to a new CodeValue is added to genv.

Top Level Expression: The expression is evaluated under genv, and the result is discarded.

Sequence of Statements: The statements are evaluated sequentially in order.

7 Understanding the Sample Programs

Read through the sample programs in the `tests` folder to familiarize yourself with how Feeny programs look and are written. And answer the following questions:

1. (4 points) The comparison operators, e.g. `1 < 2`, return either Null or 0 which is somewhat odd compared to other programming languages. Explain why Feeny might have been designed this way. How else could Feeny have been designed and what are the tradeoffs?
2. (3 points) Feeny's object model is significantly different than that of Java, Python, and Ruby. In `cplx.feeny`, explain what the function `cplx` does, and explain how it would have been written in one of the above languages.
3. (3 points) In `cplx.feeny`, note that the fields `real` and `imag` are always accessed through the slot expression `this.real` and `this.imag`. Read through the semantics of method calls in Feeny again, and explain why this is necessary. What would happen if `this.` was omitted?

4. (3 points) For `cplx.feeny`, explain what advantages the complex number implementation in Feeny has over a similar implementation in Java.

5. (4 points) Explain the significance of what is demonstrated by `inheritance.feeny`. What does Feeny's object model allow you to do that is not possible in Java?

6. (4 points) Java's specification goes into detail about the *default* values that class fields and variables have before they are initialized. For example, `int`s have default value 0, and objects have default value `null`, etc. However, there is no mention of this in the specification for Feeny. Explain why.

7. (4 points) There are two competing schools of thought in language design concerning which of *closures* or *objects* is the more fundamental construct. Explain how you might port a program written in another language with heavy use of closures to Feeny.

8 Programming in Feeny

The following exercises will help familiarize yourself with writing Feeny code. Use the supplied interpreter as detailed in section 2 to test your program.

1. (25 points) **Towers of Hanoi**: Towers of Hanoi is a classic logic puzzle that you will solve with Feeny. There are three stacks, A, B, and C, that can each hold a pile of plates. Stacks B and C are initially empty, and stack A initially holds six plates sorted with the largest one on the bottom and the smallest one on top. The only move you are allowed to make is to remove a single plate from the top of one stack and place it on the top of another, with the restriction that you can never place a larger plate on top of a smaller plate. Write a program that prints out the moves you need to move all the plates from stack A to stack B. The output should be formatted to look like this:

```
Move plate from A stack to C stack
Move plate from A stack to B stack
Move plate from C stack to B stack
Move plate from A stack to C stack
...
```

2. (25 points) **Stacks**: Here you will implement a stack library that supports the following operations:

   - Creating a stack with a given maximum capacity.
   - Pushing a new item onto the stack.
   - Peeking at the top item on the stack.
   - Popping an item from the stack.
   - Getting the current size of the stack.

   Demonstrate that your stack library is correct by making sure the following test runs:

```
var s = stack(10)

;Push items
var i = 0
while i < 10 :
   s.push(i * 10)
   i = i + 1

;Pop items
while s.size > 0 :
   printf("About to pop: ~\n", s.peek())
   printf("Popped: ~\n", s.pop())
```

3. (25 points) **More Towers of Hanoi**: Use your stack library from the previous exercise to refine your implementation of Towers of Hanoi. The initial six plates on stack A will now each be labeled with a number. The largest plate at the bottom is plate 6, and the smallest plate at the top is plate 1. For each move, in addition to printing out which stack you are moving the plate from and to, also print out which plate you are moving. The output should be formatted to look like this:

```
Move plate 1 from A stack to C stack
Move plate 2 from A stack to B stack
Move plate 1 from C stack to B stack
Move plate 3 from A stack to C stack
...
```

9 Deliverables

Students may work in pairs or alone. Please submit your answers to section 7 as *report_XX.pdf*, and your programs for section 8 as *hanoi_XX.feeny*, *stacks_XX.feeny*, and *morehanoi_XX.feeny*. Zip all files together in a file called *assign1_XX.zip*. Replace *XX* above with your initials. Mail the zip file to `patrickli.2001@gmail.com` with `[Feeny]` in the subject header.