Project #3: Brewin++ Interpreter
CS131 Fall 2023
Due date: 11/19/23, 11:59pm

Warning: Expect project #3 to take 20-30 hours of time - it's about the same amount of work as project #2!

Warning: Do not clone our GitHub repo for project #2 if you use our solution, since this will force your cloned repo to be public. If/when folks cheat off your publicly-posted code, you'll have to explain to the Dean's office why you're not guilty of cheating.

DON'T WAIT UNTIL THE LAST MINUTE TO DO THIS PROJECT!

Table of Contents

Introduction

The Brewin standards body (aka Carey) has met and has identified a bunch of new improvements to the Brewin language - so many, in fact, that they want to update the language's name to Brewin++. In this project, you will be updating your Brewin interpreter so it supports these new Brewin++ features. As before, you'll be implementing your interpreter in Python, and you may solve this problem using either your original project #2 solution, or by modifying the project #2 solution that we will provide on 11/08/23.

Once you successfully complete this project, your interpreter should be able to run syntactically-correct Brewin++ programs.

What's New in Brewin++?

You'll be adding the following features to your interpreter in project #3:

- Brewin++ now supports first-class functions (e.g., passing functions as arguments, returning functions, storing functions in variables)
- Brewin++ now supports lambdas/closures
- Brewin++ now supports pass-by-reference parameter passing in addition to pass-by-value
- Brewin++ now supports limited type coercions

Here are some examples:

First-class function support:

```
func foo(a) { print(a); }

func main() {
  a = foo;   /* store function foo in variable a */
  a(10);     /* call function foo through variable a, prints 10 */
}
```

Lambda support:

```
func main() {
  b = 5;
  f = lambda(a) {
       print(a*b);
      };   /* captures b = 5 */
  b = 10;  /* does not affect value of b captured by the lambda */
```

```
  f(20);   /* prints 100 (5 * 20) */
}
```

Pass by reference support:

```
func foo(ref a) {
  a = "abc";
}

func main() {
  b = 5;
  foo(b);
  print(b);  /* prints abc */
}
```

Type coercion:

```
func main() {
  a = 50;
  if (a*3) {   /* a*3 is coerced into a boolean value */
    print("150 is coerced to true");
  }
}
```

What Do You Need To Do For project #3?

Now that you have a flavor for the updated language, let's dive into the details.

For this project, you will create a new class called *Interpreter* within a file called *interpreterv3.py* and derive it from our InterpreterBase class (found in our provided *intbase.py*). As with project #2, your Interpreter class MUST implement at least the constructor and the run() method that is used to interpret a Brewin program, so we can test your interpreter. You may add any other public or private members that you like to your Interpreter class. You may also create other modules (e.g., variable.py) and leverage them in your solution.

Brewin++ Language Spec

The following sections provide detailed requirements for the Brewin++ language so you can implement your interpreter correctly.

First Class Functions

In Brewin++, you can perform the following first-class function operations:

- You can store functions/lambdas in variables
- You can return functions/lambdas from functions
- You can compare functions/lambdas/variables with == and != to see if they refer to the same function
- You can call functions through a variable that holds a function

Storing Functions in a Variable

To store a function in a variable, just use regular assignment and reference the function's name (e.g., foo) or lambda expression you'd like to store in the variable:

```
func foo() { ... }

func main() {
  x = foo;                                    /* assigns x to function foo */
  y = lambda() { print("I'm a lambda"); };  /* assigns y to a lambda */
}
```

You must meet the following requirements when supporting storing of functions/lambdas in variables:

- If you define overloaded functions (i.e., multiple functions called foo with different numbers of parameters), and try to assign a variable to that function name (e.g., x = foo;), this must result in an error of ErrorType.NAME_ERROR since it is ambiguous which function you are trying to store in the variable
- Assignment of one variable to another that holds a function/lambda will cause both variables to refer to the same function/lambda - no copy is made, e.g.:

```
func main() {
  x = 0;
  a = lambda() { x = x + 1; print(x); };
  b = a;
  a(); /* prints 1 */
  b(); /* prints 2 */
```

```
  a(); /* prints 3 */
  b(); /* prints 4 */
}
```

- Assignment of a variable to a named function will NOT cause any capture of in-scope variables at the time of assignment, e.g.:

```
func foo() {
  print(y);   /* generates an ErrorType.NAME_ERROR */
}

func bar() {
  y = 10;
  x = foo;    /* y is not captured by this assignment */
}

func main() {
  x = nil;
  bar();
  x();
}
```

- Note: We will never define variables that have the same name as functions, or visa-versa. Your interpreter need not handle this case and may have undefined behavior which may differ from what our canonical implementation does.
- We will also never ask you to overwrite a function with a variable using the same name, nor by reference. Your interpreter need not handle this case and may have undefined behavior which may differ from what our canonical implementation does.

Returning Functions

To return a function/closure, just use *return* followed by a function's name, a variable that holds a function, or a lambda expression:

```
func bar() { ... }

func foo() { return bar; }           /* returns function bar */
func bletch() { x = bar; return x; } /* also returns function bar */
func boop() { return lambda(q) { print(q); }; } /* returns closure */

func main() {
```

```
  x = foo();      /* assigns x to bar */
  y = bletch();   /* assigns y to bar */
  z = boop();     /* assigns z to closure */
}
```

When a function/closure is returned, a deep copy must be returned (e.g., a copy of the closure) such that any changes to the original closure don't impact the returned closure or visa-versa.

Comparing Functions

To compare functions, you may use the == and != operators. You may compare:

- a function, by its name, to another function or variable that holds a function
- a variable that holds a function/closure with another variable that holds a function/closure
- a function, by its name, with *nil*
- a variable that holds a function/closure with *nil*

Here are a few examples:

```
func foo() {
  print("hello world!");
}

func main() {
  if (foo == main) { print("wait what?"); }
  x = foo;
  if (x == foo) { print("yup"); }
  y = main;
  if (x != y) { print("that's better!"); }
  if (x != 5) { print("that's good too"); }
  if (x != nil) { print("it's not nil"); }
}
```

This prints:

yup
that's better!
that's good too
it's not nil

You must meet the following requirements when implementing function comparisons (which also includes lambda comparisons):

- Use of any comparison operator other than == and != on a variable holding a function/closure must result in an error of ErrorType.TYPE_ERROR.
- Note that a copy of a closure or function (e.g., one returned by a function, since functions return deep copies) is NOT the same as the original closure/function, so comparison using == would be false, and != would be true:

```
func foo() {
  return foo;
}

func main() {
  print(foo() == main);   /* prints false */
}
```

Calling Functions Through a Variable Holding a Function

Given a variable that holds a function/closure, you may now use that variable to make a call to the function/closure:

```
func bar(a) {
  return a * 3;
}

func main() {
  f = bar;
  x = f(10);  /* calls the bar function through variable f */
  print(x);   /* prints 30 */
}
```

You only need to handle traditional function calls via a function name, and calls through a variable that holds a function/closure. You do NOT need to handle cases like this:

```
func foo(x,y) { ... }

func bar() {
  return foo;
}
```

```
func main() {
  bar()(10,20);  /* this is a syntax error! no need to handle this */
}
```

You must meet the following requirements when implementing function calls through variables (these rules also apply to lambda functions):

- Attempting to make a function call through a variable that does NOT hold a function must result in an error of ErrorType.TYPE_ERROR, e.g.:

```
func main() {
  x = 5;    /* or x = nil; x = "abc"; x = false; etc. */
  x(10);   /* ErrorType.TYPE_ERROR */
}
```

- Attempting to make a function call through a variable with the wrong number of parameters must result in an error of ErrorType.TYPE_ERROR, e.g.:

```
func main() {
  x = main;
  x(10);       /* ErrorType.TYPE_ERROR since main() takes no args */
}
```

and:

```
func main() {
  y = lambda(x) { print(x); };
  y(10, 20);   /* ErrorType.TYPE_ERROR since lambda takes 1 arg */
}
```

Lambda Functions

Brewin++ lambda functions may have zero or more formal parameters passed by value or reference, and are defined using the following syntax:

lambda*(param1,* ***ref*** *param2, param3, …) {*
statement1;
statement2;
…

}

A lambda definition is an expression, and thus may be used in any operation that would otherwise support an expression, e.g.:

- You can assign a variable to a lambda expression, e.g.:

  x = lambda(x) { print(x); };

- You can pass a lambda expression as an argument to a function, e.g.:

  *foo(lambda(x) { return 3*x; });*

- You can compare lambdas against other functions/variables
  - A lambda function will only have equality to itself

  x = lambda() { print("Hi"); };
  if (x == x) { print("Same!"); }

- You can return a lambda expression:

  return lambda() { print("hello world!"); };

When you define a lambda function, you create a closure which captures all variables that are currently in-scope (including variables defined in other functions that are visible to the code defining the lambda via dynamic scoping).

```
func foo() {
  b = 5;
  f = lambda(a) { print(a*b); };   /* captures b = 5 */

  return f;
}

func main() {
  x = foo();
  x(20);   /* prints 100, the call to the lambda has access to b = 5 */
}
```

Here are the rules for capture:

- If a lambda has a formal parameter that is named the same as another variable that's captured, then the formal parameter shadows the captured variable. For example:

```
func main() {
  x = 5;
  y = lambda(x) { print(x); };  /* parameter x shadows captured x */
  y(10);  /* prints 10, not 5 */
}
```

- During a call to a lambda function, if the lambda captured a variable that is named the same as a variable that's currently in-scope at the time of the function call, then the captured variable shadows the in-scope variable. For example:

```
func main() {
  x = 5;
  y = lambda() { print(x); };
  x = 10;
  y();  /* prints 5 since captured x shadows in-scope x=10 */
}
```

- All variables are captured by value via deep-copy. In other words, changes to the variables by the lambda function when it runs will NOT impact the value of the original variable of the same name. Similarly, changes to the original variable after it has been captured will not affect the previously-captured value. For example:

```
func main() {
  b = 5;
  f = lambda(a) { print(a*b); }; /* captures b = 5 by making a copy */
  b = 7;                         /* has no impact on captured b */

  f(3);     /* prints 15 */
}
```

```
func main() {
  b = 5;
  f = lambda(a) { b = b + 10; };  /* modifies copy of b, not orig b */
  f(3);
  print(b);  /* prints 5 */
}
```

- If you make successive calls to a closure (created via a lambda) and the code in that closure makes changes to its captured variables, the closure will maintain those changes to the captured variables across calls to the closure, e.g.:

```
func main() {
  b = 5;
  f = lambda(a) { b = b + a; print(b); };
  f(1);     /* prints 6 */
  f(10);    /* prints 16 */
  f(4);     /* prints 20 */

  print(b); /* prints 5, original variable is unchanged */
}
```

Parameter Passing by Reference

In addition to passing parameters by value, Brewin++ now supports passing parameters by reference. In front of each formal parameter you may now add the optional "ref" keyword to indicate that the argument is passed by reference. Here's the syntax:

```
func func_name(ref param1, ref param2, ...) {
   ...
}
```

And here are a few examples of its usage showing both pass-by-reference and pass-by-value side-by-side:

```
func foo(ref x, delta) { /* x passed by reference, delta passed by value */
  x = x + delta;
  delta = 0;
}

func main() {
  a = 10;
  delta = 1;
  foo(a, delta);
  print(a);     /* prints 11 */
  print(delta); /* prints 1 */
}
```

```
func foo(f1, ref f2) {
  f1();  /* prints 1 */
  f2();  /* prints 1 */
}

func main() {
  x = 0;
  lam1 = lambda() { x = x + 1; print(x); };
  lam2 = lambda() { x = x + 1; print(x); };
  foo(lam1, lam2);
  lam1();  /* prints 1 */
  lam2();  /* prints 2 */
}
```

Returning Values

All values returned in Brewin++ must be returned "by value," in other words, a deep copy of the value must be returned. This is true for all primitive types (e.g., int, string, bool) as well as closures. Here's an example:

```
func foo(ref x) {
  return x;   /* returns a deep copy of the value x that refers to */
}

func main() {
  a = 0;
  b = foo(a);
  a = a + 1;
  print(b); /* still zero, since b is a copy of a's original value */
  b = b + 1;
  print(a); /* still 1, for the same reason */
}
```

Type Coercions

Brewin ++ now supports simple type coercions (i.e., implicit conversions). The following coercions must be implemented in your interpreter:

- When comparing an integer and a boolean (or visa-versa) using == or !=, you must coerce the integer to a boolean value of true/false before doing the comparison).
    - Integers with a value of zero are always converted to false
    - All non-zero values are converted to true

```
a = -5;
if (true == a) { print("This will print!"); }
```

- If an integer or integer expression is used anywhere a boolean value is expected, e.g.:
    - if (expression) { … }
    - while (expression) { … }
    - Boolean expressions with &&, || or logical negation (!)

  then the integer value will be coerced into a boolean before its use:

```
a = 5;
if (a) { print("This will print!"); } /* a coerced to true */
if (!0) { print("This will print!"); } /* 0 coerced to false */
if (false || 6) { print("This prints!"); } /* 6 coerced to true */
a = 3;
while (a) {  /* value of a is coerced to a boolean */
  print("This prints 3 times!");
  a = a - 1;
}
```

- When evaluating an arithmetic expression with (+, -, * or / but NOT unary negation), all boolean values are converted to integers prior to evaluating the expression:
    - A boolean value of true is always converted to 1
    - A boolean value of false is always converted to 0

```
x = true + 6; /* x is 7 */
y = false * 10; /* y is zero */
z = true + true; /* z is 2 */
```

- NO other coercions must be performed by your code (e.g., int->string, string->int), arithmetic negation of bools, etc.

Abstract Syntax Tree Spec

As in project #2, the AST will contain a bunch of nodes, represented as Python *Element* objects. You can find the definition of our *Element* class in our provided file, *element.py*. Each *Element* object contains a field called *elem_type* which indicates what type of node this is (e.g., function, lambda, statement, expression, variable, reference argument, …). Each *Element* object also holds a Python dictionary, held in a field called *dict*, that contains relevant information about the node.

Here are the different types of nodes you must handle in project #3, with changes from project #2 **bolded** for clarity:

Program Node

A *Program* node represents the overall program, and it contains a list of *Function* nodes that define all the functions in a program.

A Program Node will have the following fields:

- self.elem_type whose value is 'program', identifying this node is a *Program* node
- self.dict which holds a single key 'functions' which maps to a list of *Function Definition* nodes representing each of the functions in the program (in project #1, this will just be a single node in the list, for the main() function)

Function Definition Node

A *Function Definition* node represents an individual function, and it contains the function's name (e.g. 'main'), list of formal parameters, and the list of statements that are part of the function:

A *Function Node* will have the following fields:

- self.elem_type whose value is 'function', identifying this node is a *Function* node
- self.dict which holds three keys
    - 'name' which maps to a string containing the name of the function (e.g., 'main')
    - 'args' which maps to a list of *Argument* or ***Reference Argument*** nodes
    - 'statements' which maps to a list of Statement nodes, representing each of the statements that make up the function, in their order of execution

Lambda Definition Node

A *Lambda Definition Node* represents the definition of an anonymous function, and it contains the list of formal parameters and the list of statements that are part of the function:

A *Lambda Node* will have the following fields:

- **self.elem_type whose value is 'lambda', identifying this node is a *Lambda Node***
- **self.dict which holds two keys**
    - **'args' which maps to a list of *Argument* or *Reference Argument* nodes**
    - **'statements' which maps to a list of Statement nodes, representing each of the statements that make up the function, in their order of execution**

Argument Node

An *Argument Node* represents a formal parameter within a function definition.

An *Argument Node* will have the following fields:

- self.elem_type whose value is 'arg'
- self.dict which holds one key
    - 'name' which maps to a string holding the name of the formal parameter, e.g. 'x' in func f(x) { … }

Reference Argument Node

A *Reference Argument Node* represents a formal reference parameter within a function definition.

An *Reference Argument Node* will have the following fields:

- **self.elem_type whose value is 'refarg'**
- **self.dict which holds one key**
    - **'name' which maps to a string holding the name of the formal parameter, e.g. 'x' in func f(ref x) { … }**

Statement Node

A *Statement* node represents an individual statement (e.g., print(5+6);), and it contains the details about the specific type of statement (in project #1, this will be either an assignment or a function call):

A *Statement* node representing an assignment will have the following fields:

- self.elem_type whose value is '='
- self.dict which holds two keys
    - 'name' which maps to a string holding the name of the variable on the left-hand side of the assignment (e.g., the string 'bar' for bar = 10 + 5;)

    - 'expression' which maps to either an *Expression* node (e.g., for bar = 10+5;), a *Variable* node (e.g., for bar = bletch;) or a *Value* node (for bar = 5; or bar = "hello";)

A *Statement* node representing a function call will have the following fields:

- self.elem_type whose value is 'fcall'
- self.dict which holds two keys
    - 'name' which maps to the name of the function that is to be called in this statement (e.g., the string 'print')
    - 'args' which maps to a list containing zero or more *Expression* nodes, *Variable* nodes or *Value* nodes that represent arguments to the function call

A *Statement* node representing an "if statement" will have the following fields:

- self.elem_type whose value is 'if'
- self.dict which holds two or three keys
    - 'condition' which maps to a boolean expression, variable or constant that must be True for the if statement to be executed, e.g. x > 5 in if (x > 5) { … }
    - 'statements' which maps to a list containing one or more statement nodes which must be executed if the condition is true
    - 'else_statements' which is optional (since some if-statements don't have else clauses); this maps to a list containing one or more statement nodes which must be executed if the condition is false

A *Statement* node representing a while loop will have the following fields:

- self.elem_type whose value is 'while'
- self.dict which holds two keys
    - 'condition' which maps to a boolean expression, variable or constant that must be true for the body of the while to be executed, e.g. x > 5 in while (x > 5) { … }
    - 'statements' which maps to a list containing one or more statement nodes which must be executed if the condition is true

A *Statement* node representing a return statement will have the following fields:

- self.elem_type whose value is 'return'
- self.dict which holds zero or one key
    - 'expression' which maps to an expression, variable or constant to return (e.g., 5+x in return 5+x;)
    - If 'expression' is not in the dictionary, it means that the return statement returns a default value of nil

Expression Node

An *Expression* node represents an individual expression, and it contains the expression operation (e.g. '+', '-', **'*', '/','==', '<', '<=', '>', '>=', '!=', 'neg', '!', etc.**) and the argument(s) to the expression. There are three types of expression nodes you need to be able to interpret:

An *Expression* node representing a binary operation (e.g. 5+b) will have the following fields:

- self.elem_type whose value is any one of the binary arithmetic or comparison operators
- self.dict which holds two keys
    - 'op1' which represents the first operand the to the operator (e.g., 5 in 5+b) and maps to either another *Expression* node, a *Variable* node or a *Value* node
    - 'op2' which represents the first operand the to the operator (e.g., b in 5+b) and maps to either another *Expression* node, a *Variable* node or a *Value* node

An *Expression* node representing a unary operation (e.g. -b or !result) will have the following fields:

- self.elem_type whose value is either 'neg' for arithmetic negation or '!' for boolean negation
- self.dict which holds one key
    - 'op1' which represents the operand the to the operator (e.g., 5 in -5, or x in !x, where x is a boolean variable) and maps to either another *Expression* node, a *Variable* node or a *Value* node

An *Expression* node representing a function call (e.g. factorial(5)) will have the following fields:

- self.elem_type whose value is 'fcall'
- self.dict which holds two keys
    - 'name' which maps to the name of the function being called, e.g. 'factorial'
    - 'args' which maps to a list containing zero or more *Expression* nodes, *Variable* nodes or *Value* nodes that represent arguments to the function call

Variable Node

A *Variable* node represents an individual variable that's referred to in an expression or statement:

An *Variable* node will have the following fields:

- self.elem_type whose value is 'var'
- self.dict which holds one key
    - 'name' which maps to the variable's name (e.g., 'x')

Value Node

There are three types of *Value* nodes (representing integers, string, boolean and nil values):

An *Value* node representing an int will have the following fields:

- self.elem_type whose value is 'int'
- self.dict which holds one key
    - 'val' which maps to the integer value (e.g., 5)

A *Value* node representing a string will have the following fields:

- self.elem_type whose value is 'string'
- self.dict which holds one key
    - 'val' which maps to the string value (e.g., "this is a string")

A *Value* node representing a boolean will have the following fields:

- self.elem_type whose value is 'bool'
- self.dict which holds one key
    - 'val' which maps to a boolean value (e.g., True or False)

A *Value* node representing a nil value will have the following fields:

- self.elem_type whose value is 'nil'

Nil values are like nullptr in C++ or None in Python.

Things We Will and Won't Test You On

Your may assume the following when building your interpreter:

- WE WILL NOT TEST YOUR INTERPRETER ON SYNTAX ERRORS OF ANY TYPE
    - You may assume that all programs that we present to your interpreter will be *syntactically* well-formed and not have any syntax errors. That means:
    - There won't be any mismatched parentheses, mismatched quotes, etc.
    - All statements will be well-formed and not missing syntactic elements
    - All variable names will start with a letter or underscore (not a number)
- WE WILL TEST YOUR INTERPRETER ON ONLY THOSE SEMANTIC and RUN-TIME ERRORS EXPLICITLY SPECIFIED IN THIS SPEC
    - You must NOT assume that all programs presented to your interpreter will be *semantically* correct, and must address those errors that are explicitly called out in this specification, via a call to InterpreterBase.error() method.
    - You will NOT lose points for failing to address errors that aren't explicitly called out in this specification (but doing so might help you debug your code).
    - Examples of semantic and run-time errors include:
        - Operations on incompatible types (e.g., adding a string and an int)
        - Passing an incorrect number of parameters to a method
        - Referring to a variable or function that has not been defined

    - You may assume that the programs we test your interpreter on will have AT MOST ONE semantic or run-time error, so you don't have to worry about detecting and reporting more than one error in a program.
    - You are NOT responsible for handling things like integer overflow, integer underflow, etc. Your interpreter may behave in an undefined way if these conditions occur. Will will not test your code on these cases.
- WE WILL NOT TEST YOUR INTERPRETER ON EFFICIENCY, EXCEPT: YOUR INTERPRETER NEEDS TO COMPLETE EACH TEST CASE WITHIN 5 SECONDS
    - It's *very* unlikely that a working (even if super inefficient) interpreter takes more than one second to complete any test case; an interpreter taking more than 5 seconds is almost certainly an infinite loop.
    - Implicitly, you shouldn't have to *really* worry about efficient data structures, etc.
- WHEN WE SAY YOUR INTERPRETER MAY HAVE "UNDEFINED BEHAVIOR" IN A PARTICULAR CIRCUMSTANCE, WE MEAN IT CAN DO ANYTHING YOU LIKE AND YOU WON'T LOSE POINTS
    - Your interpreter does NOT need to behave like our Barista interpreter for cases where the spec states that your program may have undefined behavior.
    - Your interpreter can do anything it likes, including displaying pictures of dancing giraffes, crash, etc.

Coding Requirements

You MUST adhere to all of the coding requirements stated in project #1 and #2, and:

- You must name your interpreter source file *interpreterv3.py*.
- You may submit as many other supporting Python modules as you like (e.g., *statement.py*, *variable.py*, …) which are used by your *interpreterv3.py* file.
- You MUST NOT modify our intbase.py, brewlex.py, or brewparse.py files since you will NOT be turning these file in. If your code depends upon modified versions of these files, this will result in a grade of zero on this project.

Deliverables

For this project, you will turn in at least two files via GradeScope:

- Your interpreterv3.py source file
- A readme.txt indicating any known issues/bugs in your program (or, “all good!”)
- Other python source modules that you created to support your interpreterv3.py module (e.g., variable.py, type_module.py)

You MUST NOT submit intbase.py, brewparse.py or brewlex.py; we will provide our own. **You must not submit a .zip file**. On Gradescope, you can submit any number of source files when uploading the assignment; assume (for import purposes) that they all get placed into one folder together.

We will be grading your solution on **Python 3.11**. **Do not use any external libraries that are not in the Python standard library.**

Whatever you do, make sure to turn in a python script that is capable of loading and running, even if it doesn’t fully implement all of the language’s features. We will test your code against dozens of test cases, so you can get substantial credit even if you don’t implement the full language specification.

The TAs have created a template GitHub repository that contains intbase.py (and a parser bparser.py) as well as a brief description of what the deliverables should look like.

Grading

Your score will be determined entirely based on your interpreter’s ability to run Brewin programs correctly (however you get karma points for good programming style). A program that doesn’t run with our test automation framework will receive a score of 0%.

The autograder we are using, as well as a subset of the test cases, is publicly available on GitHub. Other than additional test cases, the autograder is exactly what we deploy to Gradescope. Students are encouraged to use the autograder framework and provided test cases to build their solutions. Students are also STRONGLY encouraged to come up with their own test cases to proactively test their interpreter.

We strongly encourage you to write your own test cases. The TAs have developed a tool called barista (barista.fly.dev) that lets you test any Brewin code and provide the canonical response. In discussion, TAs will discuss how to use our test infrastructure and write your own test cases.

Your score on this project will be the score associated with the final project submission you make to GradeScope.

Academic Integrity

The following activities are NOT allowed - all of the following will be considered **cheating**:

- Publishing your source code on an open GitHub repo where it might be copied (you MAY post your source code on a public repo *after* the end of the quarter)
    - Warning, if you clone a public GitHub repo, it will force the clone to also be public. So create your own private repo to avoid having your code from being used by another student!
- Leveraging ANY source code from another student who is NOW or has PREVIOUSLY been in CS131, IN ANY FORM
- Sharing of project source code with other students
- Helping other students debug their source code
- Hacking into our automated testing or Barista systems, including, but not limited to:
    - Connecting to the Internet (i.e., from your project source code)
    - Accessing the file system of our automated test framework (i.e., from your project source code)
    - Any attempts to exfiltrate private information from our testing or Barista servers, including but not limited to our private test cases
    - Any attempts to disable or modify any data or programs on our testing or Barista servers
- Leveraging source code from the Internet (including ChatGPT) without a citation comment in your source code
- Collaborating with another student to co-develop your project

The following activities ARE explicitly allowed:

- Discussing general concepts (e.g., algorithms, data structures, class design) with classmates or TAs that do not include sharing of source code

- Sharing test cases with classmates, including sharing the source code for test cases
- Including UP TO 50 TOTAL LINES OF CODE across your entire project from the Internet in your project, so long as:
    - It was not written by a current or former student of CS131
    - You include a citation in your comments:

```
# Citation: The following code was found on www.somesite.com/foo/bar
... copied code here
# End of copied code
```

  Note: You may have a TOTAL of 50 lines of code copied from the Internet, not multiple 50-line snippets of code!

- Using ChatGPT, CoPilot or similar code generation tools to generate snippets of code that are LESS THAN 10 LINES LONG from the Internet so long as you include a citation in your comments, so long as the total does not exceed 50 lines of code
- Using ChatGPT, CoPilot or similar code generation tools to generate any number of test cases, test code, etc.