Project #2: Brewin Interpreter
CS131 Fall 2023
Due date: 11/05 11:59pm

Warning: Expect project #2 to take 10-20 hours of time - it is much more work than project #1!

Warning: Do not clone our GitHub repo for project #1 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

In this project, you will be implementing a full interpreter for the Brewin language.

Once you successfully complete this project, your interpreter should be able to run any valid Brewin program and produce a result, for instance it should be able to run a program that computes a factorial recursively and prints out the result.

For this project, you may either build upon your own solution to project #1, or you may build upon our posted solution for project #1, without penalty.

What's New in Project #2

You'll be adding the following features to your interpreter in project #2:

- Support for one or more function definitions in addition to main()
  - Functions may have zero or more parameters
  - Functions may return a result
- Support for boolean variables in addition to integer and string variables
- Support for a new value called "nil" which is like None in Python or nullptr in C++
- Support for all binary integer arithmetic operations, including: +, -, *, / as well as unary negation
- Support for integer comparison operations, including: ==, !=, >, >=, <, <=
- Support for boolean comparison operations, including: ==, !=
- Support for boolean AND, OR and NOT operations: &&, ||, !
- Support for string comparison operations, including: ==, !=
- Support for string concatenation, e.g.: c = a + b;
- Support for if, if/else statements, and while loops
- Support for a new built-in function called inputs() to input strings from the user
- Brewin is a dynamically scoped language; more on that below

Here are a few simple programs in the full Brewin language.

```
func main() {
  print(fact(5));
}

func fact(n) {
  if (n <= 1) { return 1; }
  return n * fact(n-1);
}
```

The above program would print:
120

```
func main() {
 i = 3;
 while (i > 0) {
  print(i);
  i = i - 1;
 }
}
```

The above program would print:
3
2
1

What Do You Need To Do For Project #2?

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

For this project, you will create a new class called *Interpreter* within a file called *interpreterv2.py* and derive it from our InterpreterBase class (found in our provided *intbase.py*). As with project #1, 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.

Function Definitions

Every Brewin program consists of one or more functions - you may now define and call functions other than main(). This section describes the syntax of defining a function as well as the requirements you must fulfill in supporting functions in your interpreter.

Here's the syntax for defining a function, with zero or more arguments:

```
func function_name(arg1, arg2, ...)
    statement_1;
    statement_2;
    ...
    statement_n;
}
```

As you can see, Brewin is like Python in that you don't specify a type for parameters. A parameter can refer to any type of value that's passed in.

Here are a few examples showing how to define valid Brewin functions:

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

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

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

func main() {
  bar(5);
  bar("hi");
  bar(false || true);
}
```

```
func bletch(a,b,c) {
  print("The answer is: ", a+b*c);
}

func main() {
  bletch(1,3,2+4);
}
```

You must meet the following requirements when supporting functions in your interpreter.

- Every Brewin program must have at least one function called *main*. This is where execution of your Brewin program begins.
    - The main function takes no parameters; you do NOT need to check for this - all of our test cases will have a main function with no parameters
- Brewin programs may have one or more additional functions, defined above or below main
- Function names are case sensitive, so 'Foo' is different than 'foo'
- Every function must have at least one statement defined within it. You do NOT need to check for this, all of our functions that we test with will have at least one statement.
- Every brewin function may have zero or more arguments.
    - Arguments just have a name, like in Python. No types are specified for arguments.
    - Arguments must be passed by value/copy, such that a change to the argument does not change the original variable that was passed in or visa-versa
- To interpret a function, you must interpret all of its statements from top to bottom
- You must be able to support recursion via function calls
- You may overload a function, defining multiple versions of it that take different numbers of parameters

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

func foo(a,b) {
  print(a," ",b);
}

func main() {
  foo(5);
  foo(6,7);
}
```

Should print:
5
6 7

- A function may optionally return a value using the return command
    - If a function does not explicitly return a value, then it must return a default value of *nil* (nil is a value of its own type)

- If a function explicitly uses a return statement, but the return does not specify an expression/variable/value to return, then the function must return a default value of *nil*

```
func foo() {
 print("hello");
 /* no explicit return command */
}

func bar() {
  print("world");
  return;  /* no return value specified */
}

func main() {
   val = nil;
   if (foo() == val && bar() == nil) { print("this should print!"); }
}
```

Should print out:
this should print!

- A function call may be made to any function defined in a source file, whether or not the function was defined before or after the function that is calling it.
- If a program makes a function call to a function that has not been defined, then you must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error(), e.g.:

```
super().error(
    ErrorType.NAME_ERROR,
    f"Function {func_name} was found",
)
```

You may use any error message string you like. We will not check the error message in our testing.

- If a program makes a function call to a function with the wrong number of arguments (i.e., no defined function in the source file has the same number of arguments as those passed by the function call), then you must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error()

- If you define two functions with the same name and same number of parameters, this will result in undefined behavior, meaning that you don't need to address this case and it's OK if your interpreter doesn't work, behaves differently than ours, etc.

Statements

Every function consists of one or more statements. Statements in Brewin come in five forms:

- assignment statements (as with project #1)
- function call statements (as with project #1)
- **if statements**
- **while statements**
- **return statements**

function call statements

Function call statements are the same as in project #1, with a **few additions**:

- Your interpreter must support printing integers, strings and **now booleans** with your print() function.
    - **When you print booleans, the output must be either "true" or "false" (all lower case, quotes should not be included)**
- **You will not be tested on printing *nil* values. In such scenarios, your interpreter may behave in any way you like, and may act in a different way than our solution.**

if statements

if statements have the following syntax:

```
if (expression/variable/value) {
  statement1;   /* executes only if expression is true */
  statement2;
  ...
}
```

or

```
if (expression/variable/value) {
  statement1;   /* executes only if expression is true */
  statement2;
```

```
  ...
} else {
  statement1;   /* executes only if expression is false */
  statement2;
  ...
}
```

Where the expression, variable or value must evaluate to a boolean value. Each { block } is guaranteed to have one or more statements, and you don't need to check for this. if statements must always have { bracers } around its blocks. Programs may have nested if statements. The else-if construct is not supported.

For example:

```
if (f(x) > 5) {
  print(x);
  if (x < 30 && x > 10) {
    print(3*x);

}
```

or

```
if (f(x) > 0) {
  print(x);
} else {
  print(-x);
}
```

You must meet the following requirements when implementing if statements:

- If the expression/variable/value that is the condition of the if statement does not evaluate to a boolean, you must generate an error of type ErrorType.TYPE_ERROR by calling InterpreterBase.error().

while statements

while statements have the following syntax:

```
while (expression/variable/value) {
```

```
  statement1;   /* executes only if expression is true */

  statement2;
  ...
}
```

Where the expression, variable or value must evaluate to a boolean value. The { block } is guaranteed to have one or more statements, and you don't need to check for this. while statements must always have { bracers } around its blocks. Programs may have nested while loops.

For example:

```
while (f(x) > 5) {
  print(x);
}
```

or

```
while (true) {  /* infinite loop */
  print(x);
  while (x > 0) {
    if (x / 2 == 0) {
      print(x*x);
    }
    x = x - 1;
  }
}
```

You must meet the following requirements when implementing while statements:

- If the expression/variable/value that is the condition of the while statement does not evaluate to a boolean, you must generate an error of type ErrorType.TYPE_ERROR by calling InterpreterBase.error().

return statements

while statements have the following syntax:

```
return expression/variable/value;
```

or

```
return;
```

A return statement will immediately terminate the current function and return to the calling function. This includes the case where a return statement is issued within the block of an if, if/else, or while statement (with any level of nesting). This will immediately exit all nested blocks and exit the function.

If the return statement returns an expression/variable/value then this will be evaluated and a **deep copy** of the resulting value will be returned to the calling function.

A return statement in main will immediately terminate the program. Return statements make a deep copy of the integer/string/boolean value that's returned and return the copy.

For example:

```
return;         /* implicitly returns nil */
return 5*a;
return nil;
return true;
```

```
func foo(x) {
  if (x < 0) {
    print(x);
    return -x;
    print("this will not print");
  }
  print("this will not print either");
  return 5*x;
}

func main() {
  print("the positive value is ", foo(-1));
}
```

The above program will print:

-1
the positive value is 1

Expressions

Brewin now supports all of the following operators for integers:

- arithmetic binary operators: +, -, *, / (performs integer division like // in python)
- arithmetic unary operators: -
- comparison operators: ==, !=, <. <=, >, >=

  Unary integer negation has the highest precedence. Multiplication and division have higher precedence than addition and subtraction, and are evaluated from left-to right, e.g., 5 / 2 * 10 = 20. Comparison operators have lower precedence than all arithmetic operators. Hint: Our parser automatically generates ASTs which take into account precedence, so you shouldn't have to do any special handling to deal with precedence.

Brewin now supports all of the following operators for booleans:

- logical binary operators: || and &&
- logical unary operators: ! (for not)
- comparison operators: ==, !=

  Unary boolean negation has the highest precedence. && has higher precedence than ||. && and || have lower precedence than comparison and arithmetic operators. Hint: Our parser automatically generates ASTs which take into account precedence, so you shouldn't have to do any special handling to deal with precedence.

Brewin now supports all of the following operators for strings:

- binary operators: + (performs concatenation)
- comparison operators: ==, !=

Here are some examples:

```
print(true || false);  /* prints true */
print(true || false && false); /* prints true */
print(5/3);            /* prints 1 */
print(-6);             /* prints -6 */
print(!true);          /* prints false */

a = 3;
print(a > 5);          /* prints false */
```

```
print("abc"+"def");    /* prints abcdef */
```

You must meet the following requirements when supporting expressions in your interpreter.

- It is legal to compare values of different types to each other with == and !=. If two values are of different types, then must treat them as not equal. This includes comparing any value to *nil*.
- It is illegal to compare values of different types with any other comparison operator (e.g., >, <=, etc.). Doing so must result in an error of ErrorType.TYPE_ERROR.
- It is illegal to use arithmetic operations on non-integer types, with the exception of using + to concatenate strings. Doing so must result in an error of ErrorType.TYPE_ERROR.
- It is illegal to use the logical not operation on non-boolean types. Doing so must result in an error of ErrorType.TYPE_ERROR.
- Division of integers must result in a truncated integer result, like using a // b with Python
- You need not handle division by zero errors. Should such a division occur your interpreter can have undefined behavior, including behaviors that are different from our solution
- You can make a call to one or more functions defined in a Brewin program in an *expression*. The returned value from the function will be used in the expression. There is no required order of evaluation for function calls made within an expression.
- print() function can be called within expressions. The return value of print() should always be *nil*.
- **You must implement the inputs() function which inputs and returns a string as its return value. It must use our InterpreterBase.get_input() method to get input. It may take either one or no parameters**
- && and || must use strict evaluation, i.e. both arguments must be evaluated in all cases.

Variables

Variables in Brewin are unchanged from project #1. We will guarantee in our test cases that we will NEVER name a variable the same name as:

- any built-in keyword (e.g., func, if, while, return)
- any function defined in your program (e.g., we won't name any variable 'main')

Constants

Constants in Brewin are just like those in other languages.

- You can now have boolean constants of "true" and "false", all lower case, and without quotation marks

- Negative constants are supported via unary negation of positive integer values

For example:

```
a = true;
b = false;
c = -5;  /* this is negation of a positive value of 5 */
d = nil;
```

Scoping Rules

Brewin uses a dynamic scoping approach for variable scoping, which differs from the lexical scoping you are used to. In a dynamically-scoped programming language, a variable remains within scope from its point of definition until the encompassing block concludes its execution. Consequently, once a variable is defined, its accessibility extends to encompass function calls originating from that block, including functions invoked by other functions within the same block context. Here are the rules you must follow:

1. If a variable v is first initialized (and thus defined) within a block (e.g., a **function** { block }, an **if** { block } or a **while** { block }, then v will be in scope from the time it is initialized until the end of the block, including in all function calls made by code in that block. For example, the following program is valid:

```
func foo() {
   /* a, b and c are in scope here! */
   print("foo: ", a, " ", b, " ", c);   /* prints foo: 10 20 30 */
   b = b + 1;                           /* b is set to 21 */
}

func bar() {
   /* a, and b are in scope here! */
   print("bar: ", a, " ", b);   /* prints bar: 10 20 */
   c = 30;
   foo();
}

func main() {
   a = 10;
   b = 20;
   bar();
```

```
   print("main: ", a, " ", b);  /* prints bar: 10 21 */
}
```

and:

```
func main() {
   a = 10;    /* variable a's scope is the main() { block } */
   if (true) {
     a = "blah";  /* this changes a from the top-level block */
   }
   print(a);  /* prints: blah */
}
```

2. References to a variable v past the end of the block in which it was first initialized/defined must generate an error of ErrorType.NAME_ERROR. For example:

```
func main() {
  a = 5;        /* variable a's scope is the function's block */
  while (a == 5) {
      b = 10;   /* variable b's scope is the while block */
      a = 6;
  } /* variable b goes out of scope */
  print(a); /* works fine since a is still in scope, prints 6 */
  print(b); /* generates an error of ErrorType.NAME_ERROR */
} /* variable a goes out of scope */
```

and:

```
func foo() {
   c = 5;
}

func main() {
  foo();
  print(c); /* generates an error of ErrorType.NAME_ERROR */
}
```

3. A formal parameter will always shadow a variable of the same name defined in a calling function:

```
func foo(c) { /* formal parameter c shadows the c defined in main */
   print(c); /* prints 20 */
   c = 30;   /* alters the formal parameter, not the c from main */
}

func main() {
  c = 10;
  foo(20);
  print(c); /* prints 10 */
}
```

and:

```
func foo(c) { /* formal parameter c shadows the c defined in main */
   print(c);  /* prints 10 */
   c = 30;    /* alters the formal parameter, not the c from main */
}

func main() {
  c = 10;
  foo(c);
  print(c);   /* prints 10 */
}
```

You must implement your Brewin interpreter using dynamic scoping, as defined above.

Abstract Syntax Tree Spec

As in project #1, 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, statement, expression, variable, …). 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 #2, with changes from project #1 **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 'func', 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 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) { … }**

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 #2, this will be one of the following:

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 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 maps to None (when the if-statement doesn't have else clause) or 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 one key**
    - **'expression' which maps to an expression, variable or constant to return (e.g., 5+x in return 5+x;), or None (if 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:

- You must name your interpreter source file *interpreterv2.py*.
- You may submit as many other supporting Python modules as you like (e.g., *statement.py*, *variable.py*, …) which are used by your *interpreterv2.py* file.
- Try to write self-documenting code with descriptive function and variable names and use idiomatic Python code.
- You MUST NOT modify our intbase.py, brewlex.py, or brewparse.py files since you will NOT be turning these files 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 interpreterv2.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 interpreterv2.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.