Project #1: Brewin Interpreter
CS131 Fall 2023
Due date: 10/22, 11:59pm

Warning: While you'll be implementing an extremely basic interpreter in project #1, expect it to take 10 hours of time if your coding skills are rusty or you're not familiar with Python! DON'T WAIT UNTIL THE LAST MINUTE TO DO THIS PROJECT!

Table of Contents

Introduction

In this project, you will be implementing a simple interpreter for a new programming language, called Brewin. Brewin could be described as the lovechild of C++ and python, with a little EMACS lisp mixed in. You'll be implementing your interpreter in Python.

This project is the first of four - in the later CS131 projects, you'll be adding new features to your interpreter based on the concepts we learn in class.

Once you successfully complete this project, your interpreter should be able to run simple programs written in a subset of the Brewin language and produce a result, for instance it should be able to run a program that computes the result of a simple arithmetic expression and print this out.

Brewin v1 Language Introduction

What you'll implement in project #1 - let's call it Brewin v1 - is a small subset of the full Brewin language that you'll build over the quarter. The goal with this first project is to get you to build the simplest possible interpreter that does something useful. Later, once you understand the basics, you'll implement more advanced features in the Brewin language.

Here is a simple program in the Brewin language.

```
func main() {  /* a function that computes the sum of two numbers */
  first = inputi("Enter a first #: ");
  second = inputi("Enter a second #: ");
  sum = (first + second);
  print("The sum is ", sum, "!");
}
```

Assuming the user types in 10 for the first prompt and 20 for the second prompt, this program prints:

```
Enter a first #:
10
Enter a second #:
20
The sum is 30!
```

Let's distill some interesting facts about the Brewin v1 language that we can glean from our program:

- Formatting:
  - Spaces, tabs and newlines are used as separators
  - Comments use C++'s /* … */ syntax
- Functions:
  - Our program contains a single function called "main"
  - Execution starts on the very first statement inside of main() and proceeds from top to bottom
- Statements
  - Every Brewin statement ends in a semicolon
  - There are two types of statements: assignments and function calls (e.g., to the print() function)
  - Print function calls can accept any number of arguments of string or integer types, which are concatenated and printed out, followed by a newline
- Expressions:
  - The language supports addition and subtraction, with parentheses used to group arithmetic operations
  - An expression can also include a call to get an integer from the user using the inputi() function
- Variables:
  - Like Python, you don't explicitly define variables in Brewin (e.g., int a;). Instead you just assign a variable to an expression to create the variable
  - Like Python, you also don't specify types for variables in Brewin
- Constants
  - Brewin has positive integer and string constants, with strings represented using double-quotes

What Do You Need To Do For Project #1?

- You will build an Interpreter class in Python that will run Brewin v1 programs.
- Your Interpreter class must have a run() method that will be called with string that represent a syntactically-valid Brewin v1 program, e.g., '`func main() { first = inputi("Enter a first #: ");`'
- We will provide you with a parser function that can parse a syntactically correct Brewin program and generate an Abstract Syntax Tree (AST), a tree that holds your program's function(s), statements, and expressions. Your run() method can use the parser to parse the program source code, and then process the nodes of the AST to run the program
- You must be able to interpret a Brewin v1 program with a single function called main(), which will have one or more statements as shown above
- There are two types of statements you must support:
  - Assignment, e.g.:
    - x = 5;

        - y = x + 1 - 3;
        - z = (x + (1 - 3)) - y;
        - a_str = "this is a string"
        - magic_num = inputi("enter a magic number: ");
    - Printing, e.g.:
        - print(y," is 1 plus ",x);
        - print(a_str);

You do not need to support any features other than those described above. For example, you don't need to support concatenation of strings with the + operator. Nor do you need to support multiplication or division of integers. That said, if you want to get a head start, feel free to implement these things as well, so long as your program properly passes our test cases.

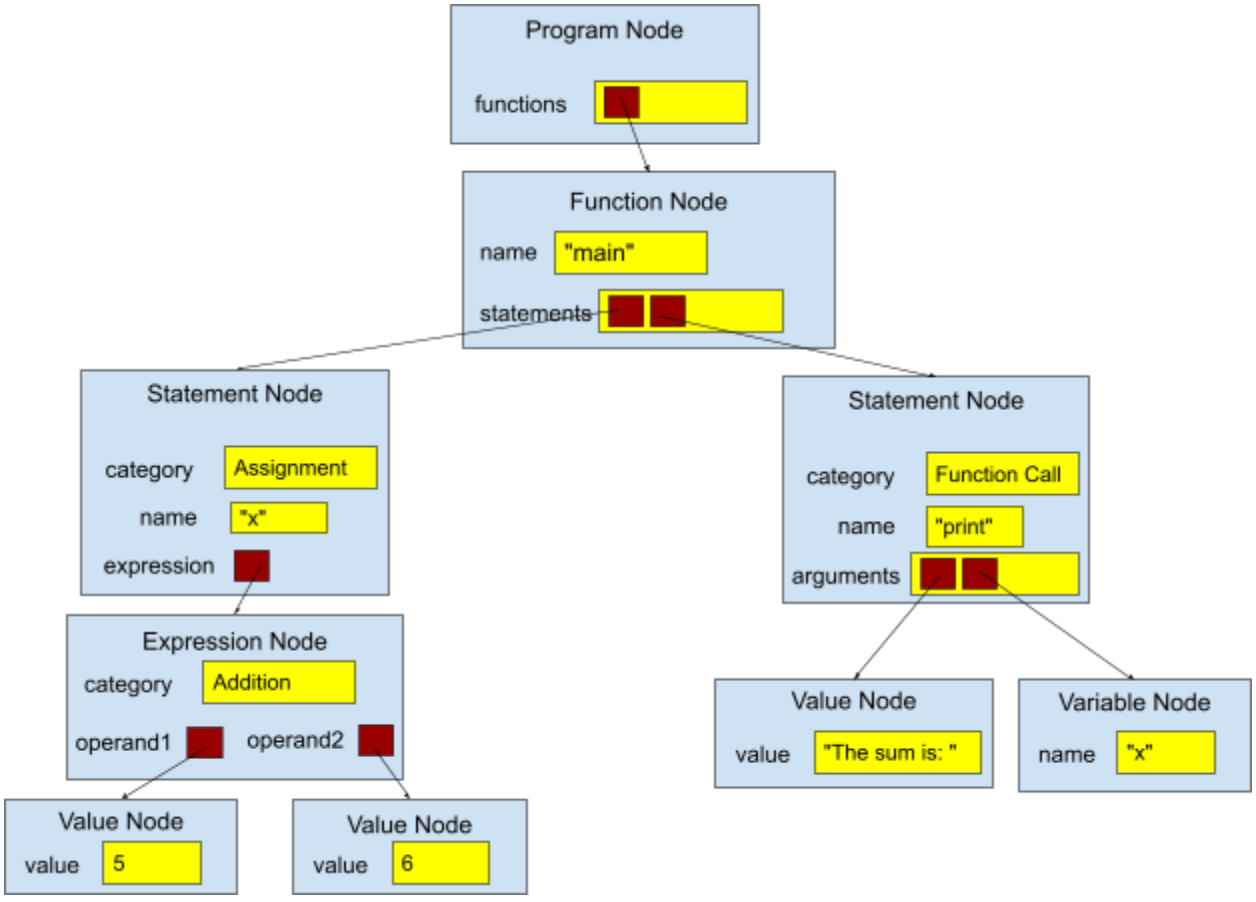
Now that you have a flavor for the language, let’s dive into the details.

How To Build an Interpreter

Before we discuss how the interpreter works, let's first introduce the concept of an Abstract Syntax Tree (AST). An Abstract Syntax Tree is a tree of nodes (like we learned about in CS32) that contains a hierarchical representation of a program. It holds one node for each different part of your program, e.g., a node for each function, each statement in each function, each expression, each value and each variable. To illustrate, consider the following Brewin program:

```
func main() {
  x = 5 + 6;
  print("The sum is: ", x);
}
```

Here's the AST for the program above:



Abstract syntax trees for Brewin have several types of nodes:

1. A Program node, which represents the overall program

    a. The program node has a field that contains a list of function nodes that make up the program
    b. In project 1, the list will contain a single function node for the main() function
    c. The program node will always be the root of the AST, since it represents the overall program
2. Function nodes, which represent individual functions
    a. Each function node has a field that holds the function's name (e.g., main(), factorial()), and another field that holds a list of statement nodes representing statements that must be run when this function is called. In project #2, we'll add parameters to functions, and so the function node will also hold a list of formal parameter names
3. Statement nodes, which represent statements
    a. Each statement node holds a field that indicates what category of statement needs to be executed (e.g., a function call or assignment)
    b. Each statement node also holds other fields that are relevant for the statement, e.g., for the statement x = 5 + 6;, the node would have two supporting fields:
        i. The first field, labeled *name*, holds a variable name, which specifies the target variable to be assigned, e.g., "x" for "x = 5 + 6;",
        ii. The second field, labeled *expression,* holds one of the following: an expression node, a variable node, or a value node, depending on whether the right side of the assignment has an expression (e.g., 5+6), a variable (e.g., "bar") or a value, e.g. 10.
4. Variable nodes, which represent variables
    a. Each variable node has a field that specifies the name of a variable, e.g. "x"
5. Value nodes, which represent values
    a. Each value node holds an integer or string value, like 52 or "hello world!"
6. An expression node, which represents binary operators like + or - (e.g., 5 + 6)
    a. Expression nodes hold two fields, op1 and op2, which refer to the two operands to the binary operator. Those fields may refer to variable nodes, value nodes or other expression nodes

Most interpreters take a source program as input (e.g., a list of strings read from the source file) and then parse this into an AST. They then interpret the AST. Here's pseudocode for how the interpreter might work, using the AST:

```
class Interpreter:
      func run(program):
            ast = parse_program(program)         # parse program into AST
            this.variable_name_to_value = Map()  # dict to hold variables
            main_func_node = get_main_func_node(ast)
            run_func(main_func_node)

      func run_func(func_node):
```

```
            for each statement_node in func_node.statements:
                  run_statement(statement_node)

      func run_statement(statement_node):
            if is_assignment(statement_node):
                  do_assignment(statement_node);
            else if is_func_call(statement_node):
                  do_func_call(statement_node);
            ...

      func do_assignment(statement_node):
            target_var_name = get_target_variable_name(statement_node)
            source_node = get_expression_node(statement_node)
            resulting_value = evaluate_expression(source_node)
            this.variable_name_to_value[target_var_name] = resulting_value

      func evaluate_expression(expression_node):
            if is_value_node(expression_node):
                return get_value(expression_node)
            else if is_variable_node(expression_node):
                return get_value_of_variable(expression_node)
            else if is_binary_operator(expression_node):
                return evaluate_binary_operator(expression_node)
            ...
      ...
```

What's going on in the pseudocode above?

1. The Interpreter is passed in a program as a list of strings that needs to be interpreted
2. The Interpreter first parses the program (using a parser module - we'll provide a parser for you) and generates an Abstract Syntax Tree (AST)
3. The interpreter creates any data structures it needs to track things like variables, e.g., a map that maps variable names to their current value (e.g., { "foo" → 11 })
7. The interpreter processes the Abstract Syntax Tree and locates the node that holds details about the main() function.
8. The interpreter then iterates through each of the statement nodes held by the main() function node and interprets each statement one after the other
    a. Depending on the statement type (e.g., an assignment vs. a function call), the interpreter runs a different method to interpret the statement
    b. As each statement is interpreted, the interpreter updates its internal state (e.g., updates a variable's value in its dictionary of variables)
    c. If a statement refers to an expression (e.g. x = 5+6;), then the expression must be evaluated to obtain a final value (e.g., 11). Similarly if a statement refers to a

variable (e.g., print(x);), then the interpreter must look up the variable name in its dictionary and obtain its current value, then execute the function call using the value

What You Need To Do

For this project, you will create a new class called *Interpreter* and derive it from our InterpreterBase class (found in our provided *intbase.py*). 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 (more details on this later in the spec). You may add any other public or private members that you like to your Interpreter class.

The above example will hopefully help you to get started on your implementation. It's just a suggestion - you may implement your interpreter in any way you like so long as it is capable of passing our test cases and meets the explicit requirements stated in this spec.

Our Parser Module

To make things easier for you, we're going to provide you with a parser module, called *brewparse.py*. Make sure that this module is in the same folder as your interpreterv1.py source file. This module holds a single function, *parse_program()*, that can take Brewin source code, passed in as a Python string, and output an Abstract Syntax Tree. This will eliminate the need for you to write your own parsing logic and let you focus on the logic of the interpreter. Here's how our parser class might be used:

```
from brewparse import parse_program     # imports parser

def main():
  # all programs will be provided to your interpreter as a python string,
  # just as shown here.
  program_source = """func main() {
                        x = 5 + 6;
                        print("The sum is: ", x);
                    }
                    """


  # this is how you use our parser to parse a valid Brewin program into
  # an AST:
  parsed_program = parse_program(program_source)
  ...
```

Abstract Syntax Tree Spec

Your 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 #1:

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') 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 two keys
    - 'name' which maps to a string containing the name of the function (e.g., 'main')
    - 'statements' which maps to a list of Statement nodes, representing each of the statements that make up the function, in their order of execution

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

Expression Node

An *Expression* node represents an individual expression, and it contains the expression operation (e.g. '+', '-') and the argument(s) to the expression. There are two 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 '+' or '-'
- self.dict which holds two keys
    - 'op1' which represents the first operand 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 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 function call (e.g. inputi("enter a number")) 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. 'inputi'
    - '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 two types of *Value* nodes (representing integers or string values):

A *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")

Brewin v1 Language Spec

The following sections provide detailed requirements for the Brewin v1 language so you can implement your interpreter correctly.

Formatting

You need not worry about program formatting (spacing, etc.) since you will use our parser code to parse all Brewin programs, and all Brewin programs that you will be tested on are guaranteed to be *syntactically* correct.

Function Definitions

Every Brewin v1 program consists of a single function, 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:

```
func function_name() {
    statement_1;
    statement_2;
    ...
    statement_n;
}
```

Here are a few examples showing how to define valid Brewin v1 functions:

```
func main() {
```

```
  a = 5 + 10;
  print(a);
  print("that's all!");
}
```

```
func main() {
  a = inputi("Enter a value: ");
  print(a);
}
```

```
func main() {
  foo = 5;
  print("The answer is: ", (10 + foo) - 6, "!");
}
```

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

- Every Brewin program must have just 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
- The main function must have at least one statement defined within it. You do NOT need to check for this, all of our main functions that we test with will have at least one statement.
- To interpret a function, you must interpret all of its statements from top to bottom
- Function names are case sensitive, so 'Main' is different than 'main'
- If a program does not have a main function defined, then you must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error(), e.g.:

  ```
  super().error(
      ErrorType.NAME_ERROR,
      "No main() function was found",
  )
  ```

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

Statements

Every function consists of one or more statements. Statements in Brewin v1 come in two forms:

- assignment statements
- function call statements

Assignment statements must have one of the following forms:

```
variable_name = expression;
variable_name = variable;
variable_name = constant;
```

Function call statements must have the following form:

```
function_name(arg_1, arg_2, …, arg_n);
```

where each argument may be an expression, variable, or a constant value. Arguments may be of either the string or integer types, or both.

Here are some example assignment statements:

```
foo = 10;
bar = 3 + foo;
bletch = 3 - (5 + 2);
prompt = "enter a number: ";
boo = inputi();
boo = inputi("Enter a number: ");
```

Here are some example function call statements:

```
print(5);
print("hello world!");
print("the answer is:", b);
print("the answer is: ", b + (a - 5), "!");
```

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

- You must support assignments
    - When performing an assignment, you must evaluate the expression/variable/value to the right of the equal sign, and then associate the variable name on the left of the equal sign with that resulting value (hint: you can use a Python dictionary for this association)

    - A variable may be assigned to values of different types during the course of execution (e.g., x = 5; x = "foo"; x = 10+3;)
- You must support function calls
    - The only function call statements you must support is the print function call and the input function call
    - Your print function call must accept zero or more arguments, which it will evaluate to get a resulting value, then concatenate without spaces into a string, and then output using the output() method in our InterpreterBase base class:

```
super().output(string_to_output)
```

    - Our output() method automatically appends a newline character after each line it prints, so you do not need to output a newline yourself.

Expressions

An expression is a combination of one or more constants, variables, operators, and/or function calls that can be evaluated to produce a single value. As with other languages, in Brewin expressions may have arbitrary nesting with parentheses.

Expressions have the following syntax:

- Arithmetic expressions in Brewin use the same as in languages like C++ and Python:
    - x + y
    - x - y
    - -x

  where x and y can be either a constant, a variable name, a function call or a nested expression which may or may not be parenthesized
- Function call expressions must have the following form:
    - function_name(arg_1, arg_2, …, arg_n)

  where the arguments can be either a constant, a variable name, a function call or an expression

Here are some sample Brewin v1 expressions:

- 3 + 5
- 4 + inputi("enter a number: ")
- 3 - (-3 + (2 + inputi()))

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

- In project #1 you must only support the addition and subtraction operations.

- Addition and subtraction expressions only need to operate on: integer constants, variables that hold integer values, calls to inputi(), and nested expressions that result in an integer value when evaluated
- Arithmetic expressions must support an arbitrary level of nesting with parentheses, e.g.,
    - ((5 + (6 - 3)) - ((2 - 3) - (1 - 7)))
- If an expression attempts to operate on a string (e.g., 5 + "foo"), then your interpreter must generate an error of type ErrorType.TYPE_ERROR by calling InterpreterBase.error(), e.g.:

```
super().error(
    ErrorType.TYPE_ERROR,
    "Incompatible types for arithmetic operation",
)
```

  You may use any error message string you like, "Incompatible types for arithmetic operation" is just an example. We will not check the error message in our testing; we will just check the error type your interpreter reports.
- In project #1, the only valid function call you can make in an *expression* is a call to the inputi() function. The inputi() function may take either no parameters or a single parameter:

  foo = inputi("Enter a number: ");
  bar = inputi();

- If an inputi() function call has a prompt parameter, you must first output it to the screen using our InterpreterBase output() method before obtaining input from the user:

```
super().output(prompt)
```

- If an inputi() expression has more than one parameter passed to it, then you must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error(), e.g.:

```
super().error(
    ErrorType.NAME_ERROR,
    f"No inputi() function found that takes > 1 parameter",
)
```

- You may assume that any given expression will only have a single call to inputi() and thus you do not need to worry about the order the input functions are called, e.g.:
    - valid: foo = inputi("enter a #: ") + 5;
    - invalid: foo = inputi("enter first #: ") + inputi("enter second #: ");

- To get input from the user when interpreting inputi(), you must call our get_input() method in our InterpreterBase base class:

```
user_input = super().get_input()
```

The get_input() method returns a string regardless of what the user types in, so you'll need to convert the result to an integer yourself. You may assume that only valid integers will be entered in response to an inputi() prompt and do NOT need to test for non-integer values being entered.

- If an expression refers to a variable that has not yet been defined, then you must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error(), e.g.:

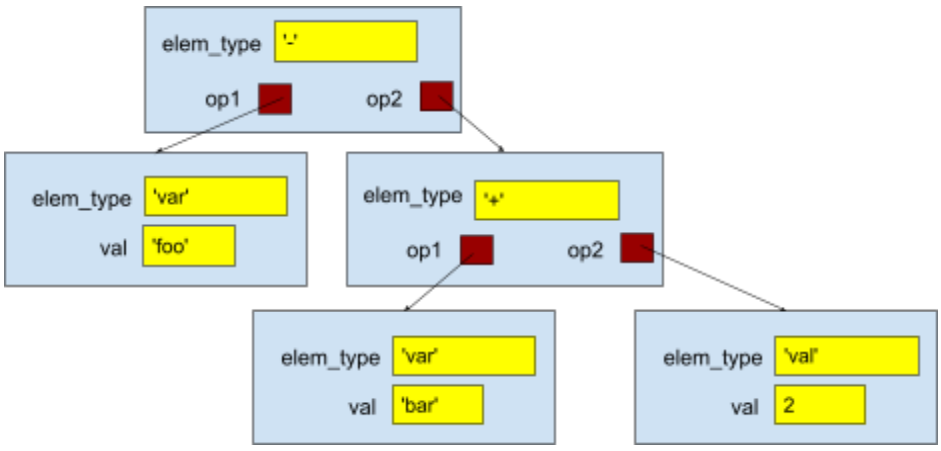
```
super().error(
    ErrorType.NAME_ERROR,
    f"Variable {var_name} has not been defined",
)
```

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

Expressions and the AST

It may be confusing at first to understand how expressions are represented in an AST. To help you understand, here's what the AST would look for the following expression:

foo - (bar + 2)



Notice that the parentheses have been removed by the parser, so all you are left with are expression, value and variable nodes. Hint: To evaluate expressions, think back to your CS32 days and consider using a post-order tree traversal.

Constants

Constants in Brewin are just like those in other languages.

- You can have integer constants, or string constants enclosed in “double quotes”
- You do not have to worry about nesting of quotes, e.g., "this is "really" bad"
- You do not need to support negative constant numbers in project #1

Here are some example constants:

- "this is a test"
- ""
- 12345678901234
- 42
- 0

Variables

Variables in Brewin are just like those in other languages.

- They may be comprised of upper-case and lower-case letters, numbers and underscores
- They must begin with an underscore or a letter
- They are case sensitive

You are NOT required to test that variable names meet the above specifications - our provided parser will take care of this.

Here are some example variable names:

- foo
- _bar_bletch
- ThisIsAVariableToo

Variables are defined upon assignment, just like in Python. For example, the following code defines the variable "x" and assigns its value to 5:

x = 5;

A variable may refer to different types of values over time. Variables in Brewin do not have fixed types. So the following is a valid program:

```
func main() {
    x = 5 + 6;
    x = "bar";
    x = 3;
}
```

You may assume that we will not test your code with variable names that are the same as functions or built-in language keywords.

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.
    - You are NOT responsible for handling unary operations for this project.

- 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 the following coding requirements for your program to work with our testing framework, and thus to get a grade above a zero on this project:

- Your Interpreter class MUST be called *Interpreter*
- You must derive your interpreter class from our *InterpreterBase* class:

```
class Interpreter(InterpreterBase):
    ...
```

- Your Interpreter class constructor (*__init__()*) must start with these two lines in order to properly initialize our base class:

```
def __init__(self, console_output=True, inp=None, trace_output=False):
    super().__init__(console_output, inp)   # call InterpreterBase's constructor
```

The *console_output* parameter indicates where an interpreted program's output should be directed. The default is to the screen. But when we run our test scripts, we'll redirect the output for evaluation. You can just pass this field onto *InterpreterBase's* constructor and otherwise ignore it.

The inp parameter is used for our testing scripts. It's the way we pass input to your program in an automated manner. You can just pass this field onto InterpreterBase's constructor and otherwise ignore it.

The *trace_output* parameter is used to help in your debugging. If the user passes True in for this, your program should output whatever debug information you think would be useful to the screen via Python's print() function. We will not test you on this, but it may be helpful for your own debugging. For example:

```
class Interpreter(InterpreterBase):
    ...
    def interpret_statement(self, statement):
        if self.trace_output == True:
            print(statement)
        ... # your code to interpret the passed-in statement
```

- You must implement a run() method in your Interpreter class. It must have the following signature:

```
def run(self, program):
    ...
```

Where the second parameter, *program*, is an array of strings containing the Brewin program text that we want to interpret, e.g.:

```
program = """func main() {
             x = 5 + 6;
             print("The sum is: ", x);
          }"""

interpreter = Interpreter()
interpreter.run(program)
```

- When you print output (e.g., *print("blah")*) your interpreter must use the InterpreterBase.output() method to ensure that your program's output can be evaluated by our test scripts. By default, our output() method will display all output to the screen, but during automated grading we will redirect the output for testing. **YOU MUST NOT USE python's print() statement to print your program's output as we will not process this data in our testing framework.**
- When you get input from the user (e.g., with an *inputi()* call) your interpreter must use the InterpreterBase.get_input() method, to ensure that our test input can be passed in by our test scripts. By default, our get_input method will prompt the user via the keyboard. **YOU MUST NOT USE python's input() function to get input from the user as it will not work with our testing framework.**
- To report errors (e.g., typing errors, name errors) you must call the InterpreterBase.error() method with the specified error:

```
def interpret_statement(self, statement):
   ...
   if cant_find_variable(v):
       super().error(ErrorType.NAME_ERROR,
                     f"Unknown variable {v}")
```

- You must name your interpreter source file *interpreterv1.py*.
- You may submit as many other supporting Python modules as you like (e.g., *statement.py*, *variable.py*, …) which are used by your *interpreterv1.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 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 interpreterv1.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 interpreterv1.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 (https://barista-f23.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.