Project #1: Brewin Interpreter
CS131 Spring 2023
Due date: May 7th, 11:59pm

Introduction

In this project, you will be implementing a simple interpreter for a new programming language, called Brewin. Brewin is an object-oriented variant of the LISP language. You guessed it - that means there are lots of parentheses! :^) You'll be implementing your interpreter in Python. This project is the first of three - 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 Brewin programs and produce a result, for instance it should be able to run a program that computes the factorial of a number the user enters (see below).

Brewin v1 Language Introduction

In Brewin, all functions and data are contained in classes - there are no stand-alone functions or variables. Every Brewin program must have a special class called "main", and this class must hold a special method called "main" where execution of your program begins.

Here is a simple program in the Brewin language. You can see our main class, with two fields (num, result) and two methods (main, factorial). Line numbers have been added for clarity:

```
000 # Our first Brewin program!
001 (class main
002  # private member fields
003  (field num 0)
004  (field result 1)
005
006  # public methods
007  (method main ()
008   (begin
009    (print "Enter a number: ")
010    (inputi num)
011    (print num " factorial is " (call me factorial num))
012   )
013  )
014 (method factorial (n)
015  (begin
016   (set result 1)
017   (while (> n 0)
018    (begin
019     (set result (* n result))
020     (set n (- n 1))
021    )
022   )
023   (return result)
024  )
025 )
026)
```

Here's a description of the above program:

- 000: This line has a comment on it, indicated by the #
- 001: Defines a class called main; all Brewin programs must have a main class. When the Brewin interpreter runs a program, it first instantiates an object of the main class's type and then executes the main method within this object.

- 003: This defines a member variable (a "field") named num whose initial value is 0. In Brewin v1, fields are not given fixed types, like in C++. A given field may thus refer to values of different types over time.
- 004: This defines a member variable (a "field") named result whose initial value is 1. All Brewin fields must have an initial value specified for them.
- 007: This defines the "main" method of our main class. Execution of a Brewin program starts with main method in the main class.
- 008: This is a begin statement. A begin statement allows you to sequence one or more nested statements (e.g., those on lines 9-11). By default, all Brewin methods are made up of a single, high-level statement, so if you want a function to execute multiple statements, a method's high-level statement must be a begin statement, which then may contain one or more embedded statements to execute.
- 009: This statement prints data to the screen.
- 010: This statement inputs an integer (hence "inputi") from the keyboard and stores it into the member variable named num.
- 011: This statement prints something like "5 factorial is 120" to the screen. To get the factorial result, it makes a method call to the factorial method within the "me" object. "me" is like "this" in C++ or "self" in Python - it refers to the current object, so this method call invokes the factorial method in the current object.
- 014: This defines the "factorial" method in our main class. You can see that the factorial method accepts one parameter called n. Parameters in Brewin v1 don't have types specified for them, meaning that n could refer to different types of values depending on what values are passed in over time.
- 015: The begin statement lets us run multiple statements in the factorial function.
- 016: The set command sets the value of the member variable result to 1.
- 017: This is a while loop which will continue running while n > 0.
- 018: The begin statement lets us run multiple statements in the while loop. The body of a while loop must be a single statement, so adding a begin statement lets us run multiple sub-statements in the while loop.
- 019: Sets the value of the result member variable to n * result.
- 020: Decrements the value of the member variable n.
- 023: Returns the value of the result member variable as the result of the factorial function.

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

- Formatting:
    - Brewin requires that statements be enclosed in parentheses like LISP. You must have the right number of parentheses or your program won't work! Too many or too few can cause problems, so be careful!
    - Comments in Brewin begin with a # sign.
    - Spaces and newlines are used as separators - there are no parentheses, commas, semicolons, etc.

- Classes:
    - Our program contains a single class called "main" (which is mandatory in all Brewin programs), but a Brewin program can have additional classes if you like.
    - Before execution begins, the Interpreter instantiates a "main" object based on the definition of the main class.
    - Once the main object has been instantiated, the Interpreter then asks that object to run the statement(s) in its main method - this executes the program.
- Variables:
    - There are two types of variables in Brewin v1 - member variables (aka fields) in classes, and parameters of functions. There are no local variables in Brewin v1, other than parameters.
    - Fields (aka member variables)
        - Fields are defined with a "field" keyword in a class.
        - You don't specify types for field variables. So field variables don't have fixed types, but the values they refer to do have types. Thus a member variable may refer to values of different types over time, just like in Python.
        - Fields can have their value changed.
        - All fields must have a constant value specified for their initial value.
        - All fields in an object are private and may only be accessed by methods in the same object
    - Parameter variables
        - Parameter variables are defined in a method's prototype.
        - Parameter variables are typeless (so you can pass different types of values each time you call the function if you like), just like in Python.
        - Parameter variables can have their value changed.
    - All variables are assigned via the "set" statement.
- Methods:
    - All methods within a class are defined via the "method" keyword.
    - All methods in a class are public.
    - The main method, within the main class, is a special method where execution of a Brewin v1 program begins.
    - Methods may have zero or more parameters, which are specified in parentheses right after the method name. So the main method has no parameters, and the factorial method has a single parameter called n.
    - Every method must have a single top-level statement as its body, e.g., (print "hello world"). If you want to run more than one statement in a method, you must make the top-level statement in the method a "begin" statement which can have one or more sequenced sub-statements, e.g.:

```
# every method has a single statement
(method talk () (print "hi there"))

(method greeting ()
```

```
  (begin    # the method's single statement is a begin statement
    # we can then sequence multiple sub-statements in the begin
    (print "hello ")
    (print "world")
  )
)
```

    - All methods are called with the "call" keyword. When you use the call keyword, you must specify what object the call is being directed to, then the method to be called in that object, then any parameters that are to be passed to the method. So the code:

      ```
      (call me factorial num)
      ```

      calls the factorial method within the me object (the current object - think of "me" like the "this" pointer in C++), passing a parameter value of num to the method. As you can see, methods may modify parameters.
    - As you can see, a method may optionally return a value.
- Expressions:
    - Expressions are written in prefix notation, e.g., (+ 5 (* 6 3)), or (== n 1).
    - An expression may refer to constants, variables and other sub-expressions, and it may call methods within the current object or other objects.
    - An expression may also be used to instantiate a new object (we'll see this later)
- Control Flow:
    - A while loop has a boolean expression as its condition (which it uses to decide whether to run its body) and a single statement as its body. To sequence multiple statements in a while loop, wrap them in a top-level “begin” statement.
    - The begin statement lets you run one or more sub-statements in order.
    - The return statement lets a method return a value to its caller, and immediately terminates the current method.

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

How To Build an Interpreter

When you build an interpreter, you have to represent all of the elements of the interpreted language (e.g., Brewin) using classes/objects in the "host" language that you're using to build the interpreter (e.g., Python). For instance, if our interpreted language:

- has variables, we'll need to have a class in the host language to represent variables (their name, their current value).
- has values, we'll need to have a class in the host language to represent a value (its type and the current value it's set to)
- has functions, we'll likely have a class in the host language to represent a function (its name, parameter variables, and statement(s) that make up its body).
- has statements, then we may want to have class(es) in the host language to represent the different types of statements in the interpreted language ("while" statements, "if" statements, "set" statements, "return" statements, etc.)
- has classes, then we'll likely have a class in the host language to represent a class in the interpreted language (holding the class's name, its fields and their initial values and its methods).
- has objects (instantiated from classes), then we'll likely have a class in the host language to represent instantiated objects (e.g., what fields/member variables the object has and what their current values are, what methods does the object have, etc.).

As your interpreter interprets a Brewin program, if the interpreted program defines a new Brewin class, then your python program will need to create a new python object to represent/track that class. If a statement in your interpreted program instantiates a new Brewin object, then your python program will need to create and track a new python object that represents that Brewin object and the methods/fields it holds. If a statement in your interpreted program defines a new variable, then your python program will need to create and track a new python object that represents that Brewin variable. And so on.

We'll also need to have an overall Interpreter class in the host language (python) that can use all of the classes above to implement the interpreter. For this project, you MUST 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.

How should your Interpreter class's run() method work? Here's some pseudocode:

```
class Interpreter(InterpreterBase):
  def run(self, program):
    # parse the program into a more easily processed form
    result, parsed_program = BParser.parse(program_source)
```

```
          if result == False:
           return # error
         self.__discover_all_classes_and_track_them(parsed_program)
         class_def = self.__find_definition_for_class("main")
         obj = class_def.instantiate_object()
         obj.run_method("main")
```

And what might a ClassDefinition class (which represents a Brewin class) look like?

```
class ClassDefinition:
  # constructor for a ClassDefinition
  def __init__(self, …):
     …

  # uses the definition of a class to create and return an instance of it
  def instantiate_object(self):
    obj = ObjectDefinition()
    for method in self.my_methods:
      obj.add_method(method)
    for field in self.my_fields:
      obj.add_field(field.name(), field.initial_value())
    return obj
```

And what might an ObjectDefinition class (which represents a Brewin object, which may be instantiated from a Brewin class during interpretation of a Brewin program) look like?

```
class ObjectDefinition:
  def __init__(self, …):
    …

  # Interpret the specified method using the provided parameters
  def call_method(self, method_name, parameters):
    method = self.__find_method(method_name)
    statement = method.get_top_level_statement()
    result = self.__run_statement(statement)
    return result

  # runs/interprets the passed-in statement until completion and
  # gets the result, if any
  def __run_statement(self, statement):
   if is_a_print_statement(statement):
     result = self.__execute_print_statement(statement)
   elif is_an_input_statement(statement):
```

```
    result = self.__execute_input_statement(statement)
  elif is_a_call_statement(statement):
    result = self.__execute_call_statement(statement)
  elif is_a_while_statement(statement):
    result = self.__execute_while_statement(statement)
  elif is_an_if_statement(statement):
    result = self.__execute_if_statement(statement)
  elif is_a_return_statement(statement):
    result = self.__execute_return_statement(statement)
  elif is_a_begin_statement(statement):
    result = self.__execute_all_sub_statements_of_begin_statement(statement)
  …
  return result
```

The above examples will hopefully help you to get started on your implementation. These are just suggestions - 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 Class

To make things easier for you, we're going to provide you with a simple class, called BParser, that can take Brewin source code, passed in as a Python list of strings, and output a fully-parsed list of strings. 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 may be used:

```
from bparser import BParser     # imports BParser class from bparser.py

def main():
  # all programs will be provided to your interpreter as a list of
  # python strings, just as shown here.
  program_source = ['(class main',
                    ' (method main ()',
                    '   (print "hello world!")',
                    ' ) # end of method',
                    ') # end of class']

  # this is how you use our BParser class to parse a valid
  # Brewin program into python list format.
  result, parsed_program = BParser.parse(program_source)
  if result == True:
    print(parsed_program)
  else:
    print('Parsing failed. There must have been a mismatched
parenthesis.')

main()
```

The above program would print the following out:

```
[['class', 'main', ['method', 'main', [], ['print', '"hello world!"']]]]
```

Notice that our parser removes extraneous spaces and newlines, and eliminates comments. You're left with a simple python list of lists that can be easily processed to build your interpreter. e.g.,

```
for class_def in parsed_program:
  for item in class_def:
    if item[0] == 'field':
      # handle a field
    elif item[0] == 'method':
      # handle a method
...
```

In addition, our parser attaches a new member variable to every string in the output list called line_num. You can use this to determine what line number each token was found on in the input program. For example, here's a function that prints the line numbers of all the tokens in the above program:

```
def print_line_nums(parsed_program):
  for item in parsed_program:
    if type(item) is not list:
      print(f'{item} was found on line {item.line_num}')
    else:
      print_line_nums(item)
```

For our above program, this would print out:

```
class was found on line 0
main was found on line 0
method was found on line 1
main was found on line 1
print was found on line 2
"hello world!" was found on line 2
```

You can use the line_num field to output specific error messages (e.g., "trying to dereference a null object reference on line 5.") which will help with debugging.

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 BParser class to parse all Brewin programs
  - All Brewin programs are guaranteed to be formatted correctly and *syntactically* correct

Classes

Every Brewin program consists of one or more classes. This section describes the syntax of defining a class as well as the requirements you must fulfill in supporting classes in your interpreter.

Overall Class Definition

Class Definition Syntax

Here's the syntax for defining a class:

```
(class classname
  (field ...)      # specific syntax will follow later in this doc
  (method ...)     # specific syntax will follow later in this doc
  ...
  (field ...)
  ...
  (method ...)
)   # end of class definition
```

Class Definition Requirements

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

- Every Brewin program must have at least one class called main, with at least one method named main in that class. This is where execution of your Brewin program begins.

- Brewin programs may have zero or more additional classes beyond the main class
- Every Brewin class must have at least one method defined within it
- Every Brewin class may have zero or more fields defined within it
- Class names are case sensitive, and must start with an underscore or letter, and may have underscores, letters and numbers
- Classes may be defined in any order within your source file; all classes are visible to all other classes (e.g., for instantiating a new object) regardless of whether they're define above or below the location of instantiation
- Methods and fields may be defined in any order within the class; all methods and fields are visible to all other methods inside the class regardless of the order they are defined
- There are no official constructors or destructors in Brewin. If you want to define and call your own methods in a class to perform initialization or destruction you may.
- Duplicate class names are not allowed. If a program defines two or more classes with the same name you must generate an error of type ErrorType.TYPE_ERROR by calling InterpreterBase.error().

Class Definition Examples

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

```
(class person
   (field name "")
   (field age 0)
   (method init (n a)
      (begin
        (set name n)
        (set age a)
      )
   )
   (method talk (to_whom)
      (print name " says hello to " to_whom)
   )
)

(class main
 (field p null)
 (method tell_joke (to_whom)
    (print "Hey " to_whom ", knock knock!")
 )
 (method main ()
   (begin
      (call me tell_joke "Matt") # call tell_joke in current object
```

```
        (set p (new person))  # allocate a new person obj, point p at it
        (call p init "Siddarth" 25) # call init in object pointed to by p
        (call p talk "Paul")       # call talk in object pointed to by p
      )
    )
  )
```

Class Fields

This section details how to define fields (i.e., member variables) inside of classes.

Field Syntax

Here's the syntax for defining a field:

```
(field field_name initial_value)
```

Field Requirements

- Field names are case sensitive and must start with an underscore or letter, and contain letters, underscores and numbers.
- All fields in Brewin are private by default and may only be accessed by methods in the same object. An object cannot access the fields of another object, even if the objects are both of the same type/class.
- Fields are visible to all methods within the current object, whether those methods are defined above or below the field's definition, just like in C++.
- Fields do not have a particular type - they can be assigned to a value of any type, and may be reassigned to values of different types over time.
- All field definitions must specify an initial value for the field. Valid initial values must be constants: integers like 10 or -20, true, false, null, and "strings" in double quotation marks. Expressions are not allowed to be used for the initial value of a field.
- When an object is instantiated in your program from a class, the object gets a copy of each of the fields defined in the class, with each field's value initialized to the specified initial value for that field.
- If a given class has two or more fields with the same name, you must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error().
- None of the programs we test you with will use “me” as the name of a field. You can treat that as undefined behavior.

Field Examples

Here are some examples of field definitions:

```
(field foo_123 10)
(field name "unknown")
(field _awesome true)
(field obj_ref_field_puppy null)
```

Class Methods

This section details how to define methods (i.e., member functions) inside of classes.

Defining Methods

Method Definition Syntax

Here is the syntax for defining a method within a class:

```
(method method_name (param_name1 param_name2 ... param_namen)
   (statement_to_execute)
) # end of method
```

Where there may be zero or more parameters to a function. Notice that in Brewin v1, you do not specify the types of the parameters or the return type of the method. Each method has a single high-level statement that it runs.

Method Definition Requirements

- Every class must have at least one method
- Each method definition specifies the method's name, its list of parameters - if any, and a single, top-level statement to run for the method
- All formal parameters to a function must have unique names, i.e. two parameters to a function cannot have the same name. You can treat this as undefined behavior.
- We will not test you on methods with formal parameters named “me”. You can treat this as undefined behavior.
- Methods in Brewin v1 do not have types for the parameters nor do they have return types. So any type of variable can be passed to a method's parameter, and any type of value may be returned (including no value at all).
- All methods are public in Bruin - there are no private methods
- Methods may optionally return a value
- If a method wishes to run multiple statements, then you must specify a begin statement as the method's top-level statement. You may then have any number of sub-statements run as part of the begin statement.

- If a method has a parameter whose name is the same as the name of a field in the class, then all references to the name of that variable during the execution of that method will refer to the parameter rather than the field (i.e., the parameter hides the field within that method). This is called shadowing. For example:

```
(class main
  (field x 10)
  (method bar (x) (print x))  # prints 5
  (method main () (call me bar 5))
)
```

- If a given class has two or more methods with the same name, you must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error().

Method Definition Examples

Here are a few example method definitions:

```
(method say_hello () (print "hello!"))          # takes no parameters
(method say_hi_to (name) (print "hello " name)) # takes 1 parameter
(method square (val) (return (* val val)))      # returns a result
(method do_several_things ()        # begin with multiple statements
   (begin
     (print "hi")
     (print "bye")
   )
)
(method two_params (a b) (return (+ a b)))
```

Calling Methods

Syntax

Method calls use the following syntax:

```
(call target_object method_name param1 ... paramn)
```

where target_object can be a member variable that holds an object reference or the "me" keyword, which indicates that the call should be directed to a method in the current object that's making the call.

Requirements

- You may pass any variable, constant or expression as parameters to a method call. For example, this would be a valid function call if some_method accepts a single parameter:

```
(call some_object some_method (* 3 (+ 5 6)))   # 33 passed as 1st arg
```

- If the called method returned a value, then you may use this return value within a statement or expression, e.g.:

```
# use the result of the method call to update field_x
(set field_x (call some_object some_method (* 3 (+ 5 6))))
```

- Brewin v1 does not support overloaded methods (e.g., two methods with a different set of parameters). You are NOT responsible for supporting overloading or detecting invalid use of overloading, and will not be provided with any test cases that use overloaded methods.
- If a call is made to an undefined method, you must report an error by calling the InterpreterBase.error() method with an error type of ErrorType.NAME_ERROR.
- If a call is made to a method with the wrong number of parameters, you must report an error by calling the InterpreterBase.error() method with an error type of ErrorType.TYPE_ERROR.

Examples

The following code shows several method calls within the same and to different objects:

```
(class person
   (field name "")
   (field age 0)
   (method init (n a) (begin (set name n) (set age a)))
   (method talk (to_whom) (print name " says hello to " to_whom))
   (method get_age () (return age))
)

(class main
 (field p null)
 (method tell_joke (to_whom) (print "Hey " to_whom ", knock knock!"))
 (method main ()
   (begin
      (call me tell_joke "Leia")  # calling method in the current obj
      (set p (new person))
```

```
        (call p init "Siddarth" 25)  # calling method in other object
        (call p talk "Boyan")        # calling method in other object
        (print "Siddarth's age is " (call p get_age))
    )
  )
)
```

Constants

Constants in Brewin are just like those in other languages.

- You can have integer constants, string constants enclosed in “double quotes”, boolean constants (true/false), and null (which is used to designate an invalid object reference). There are no floating-point numbers.
- Integer constants may be positive or negative (e.g., -5 with no spacing between the minus sign and the first digit), and have any valid integer value representable by a python integer.

Here are some example constants:

- "this is a test"
- -12345678901234
- 42
- true
- false
- null
- 0

Expressions

An expression is made up of constants, variables, and operations (e.g., +, -, <, ==, function calls, object instantiation) which yield a result/output value. An expression may have nested sub-expressions. An expression is NOT a statement! For example:

```
(+ 5 variable)
(> str "foobar")
(== some_int (+ (* 3 5) 6))
```

Expressions may be any of the following:

- A constant (e.g., 5, "foobar", true/false, null)
- A variable name (e.g., years_old)
- An arithmetic expression, with any of the following operators: +, -, *, /, %, e.g.:
    - `(set var (+ var 15))`

  The arithmetic operators must yield an integer result
- A comparison expression that compares integers with any of the following operators: <, >, <=, >=, !=, ==:
    - `(if (> some_int_var1 5) (print "bigger"))`

  The comparison operators must yield a boolean result
- A concatenation expression that concatenates two strings and results in a new string, using the + operator:
    - `(set var (+ "foo" some_var_that_refers_to_a_string))`
- A comparison expression that compares strings, with any of the following operators: ==, !=, <, >, <=, >=, e.g.:
    - `(if (>= string_var1 "foobar") (print "bigger or equal"))`

  All comparison operators compare strings lexicographically

  The comparison operators must yield a boolean result
- A comparison expression that compares booleans, with any of the following operators: !=, ==, & (logical AND), | (logical OR), e.g.:
    - `(if (== boolean_var1 true) (print "it's true"))`

  The comparison operators must yield a boolean result
- A comparison expression that compares objects with null, with the following operators: ==, !=, e.g.:
    - `(if (== x null) (print “it’s true”))`

  We will always compare with null, not another object.
- A boolean unary NOT expression, with the ! operator, e.g.:
    - `(if (! boolean_var1) (print "the variable is false"))`

  The unary NOT operator must yield a boolean result
- A method call expression that calls a method in the current object or another object, e.g.:
    - `(set result (call me factorial num))`
    - `(set result (call some_other_obj some_other_member_func 10))`

- A new expression which instantiates a new object of the specified class's type and returns an object reference to that new object, e.g.:
    - `(set some_field (new ClassName))`

  Note: In Brewin, there is no *delete* command like in C++. Brewin may rely upon Python's garbage collection feature to automatically reclaim objects that are no longer referenced, so there's no need to implement your own garbage collection.

Here are requirements for expressions:

- All expressions are represented in prefix notation with spaces separating each token in the expression, e.g., (+ 5 6), (> a 10) or (call target_object method_name param)
- The types of all values used in an expression must be compatible, e.g., you can't add a string and an integer, or compare a boolean and an integer. If the types do not match, you must generate an error by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.
- If an operator is not compatible with an operand, then you must generate an error by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.
- If an expression refers to a class name, field or parameter that does not exist, then you must generate an error by calling InterpreterBase.error() with a type error of ErrorType.NAME_ERROR.

call expression details

You may include a method call within an expression. It's usage is as follows:

```
(call target_object method_name arg1 arg2 … argn)
```

Where there may be zero or more arguments, depending on the method being called. So for example, the call expression could be used as follows:

```
(print num " factorial is " (call me factorial num))
```

For detailed requirements for method calls, please see the *Calling Methods* section above and the *call statement* section later in this spec.

new expression details

The new expression instantiates a new object of the specified class type, and returns an object reference to that object. It's usage is as follows:

```
(new class_name)
```

For example:

```
(class main
  (field other null)
  (method main ()
    (begin
      (set other (new other_class))  # HERE
      (call other foo 5 6)
    )
  )
)

(class other_class
  (field a 10)
  (method foo (q r) (print (+ a (+ q r))))
)
```

In the above example, the line marked HERE instantiates a new other_class object using the new expression, and then sets the other field to refer/point to the new object.

Requirements:

- The new expression must be able to instantiate any class in the program, whether it is defined before or after the method that performs the new expression. See the example above, where the definition of other_class is defined below the (new other_class) expression. This is legal.
- If the class name is unknown, you must generate an error of type ErrorType.TYPE_ERROR by calling InterpreterBase.error().

Statements

The sections below detail the types of statements that you must support in your interpreter.

begin statement

The begin statement lets you execute one or more sub-statements in sequence. Since method bodies, if statements and while loops in Brewin are only allowed to have a single top-level statement, if you want them to run multiple statements, you have to use a begin statement as their top-level statement and then add two or more sub-statements as you like inside the begin statement.

- Syntax:

```
(begin
  (statement1)
  (statement2)
  ...
  (statementn)
)
```

- Example usage:

```
(class main
  (field x 0)
  (method main ()
    (begin
      (print "hello")
      (print "world")
      (print "goodbye")
    )
  )
)

(class main
  (field x 0)
  (method main ()
    (if (== x 0)
      (begin            # execute both print statements if x is zero
        (print "a")
        (print "b")
      )
```

```
      )
    )
  )
```

- Requirements:
  - A begin statement must have one or more nested statements inside of it

call statement (and call expression)

The call statement performs a function call to a method within the current object or to a method in another object that is referenced by an object reference variable. All parameters are passed by value, so changes made to the parameters in the called method must make no changes to variables passed from the current method. If the called method returns a value, then the call may be used inside an expression.

- Syntax:

```
(call me method_name param1 ... paramn)     # call method in cur object
(call obj_ref method_name param1 ... paramn)# call method in other
object
```

- Example usage:

```
(class main
  (field other null)
  (field result 0)
  (method main ()
    (begin
      (call me foo 10 20)   # call foo method in same object
      (set other (new other_class))
      (call other foo 5 6)  # call foo method in other object
      (print "square: " (call other square 10)) # call expression
    )
  )
  (method foo (a b)
   (print a b)
  )
)

(class other_class
  (method foo (q r) (print q r))
  (method square (q) (return (* q q)))
)
```

- Requirements:
  - A call statement must always specify a valid object reference or the "me" keyword, which is like "self" or "this" and refers to the current object
  - Call statements may be made to any method in the same object, whether the method is defined above or below the call statement.
  - Call statements to another object may be made to a method of any class, whether the other class is defined above or below the call statement.
  - There may be zero or more arguments passed in a call statement, and all arguments must be placed directly after the method name.
  - Arguments to a call may be constants, variables or expressions.
  - All arguments are passed by value to the called method.
  - If the called function returns a value, then the call will evaluate to that return value and its result can be used within an expression.
  - A call made to an object reference of null must generate an error of type ErrorType.FAULT_ERROR by calling InterpreterBase.error().
  - A call made to a method name that is not defined for the target object must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error().
  - A call made to a method with the wrong number of parameters must generate an error of type ErrorType.TYPE_ERROR by calling InterpreterBase.error().

if statement

Used to perform an if or if/else operation.

- Syntax - there are two variations:

```
(if expression (run_if_expr_is_true))
(if expression (run_if_expr_is_true) (run_if_expr_is_false))
```

- Example usage:

```
(class main
  (field x 0)
  (method main ()
    (begin
     (inputi x)   # input value from user, store in x variable
     (if (== 0 (% x 2))
       (print "x is even")
       (print "x is odd")   # else clause
     )
     (if (== x 7)
```

```
        (print "lucky seven")  # no else clause in this version
      )
      (if true (print "that's true") (print "this won't print"))
    )
  )
)
```

- Requirements:
  - You must make the if statement works like an if or if/else statement in any language
  - If the condition of the if statement does not evaluate to a boolean type, you must generate an error of type ErrorType.TYPE_ERROR by calling InterpreterBase.error().

inputi and inputs statements

Used to input either an integer (inputi) or a string (inputs) from the user or automated testing framework.

- Syntax:

```
(inputi variable)    # sets variable to the user-provided integer value
(inputs variable)    # sets variable to the user-provided string value
```

- Example usage:

```
(class main
  (field x 0)
  (method main ()
    (begin
     (inputi x)   # input value from user, store in x variable
     (print "the user typed in " x)
    )
  )
)
```

- Requirements:
  - Your implementation MUST use our InterpreterBase.get_input() method to read input from the user (so we can do automated testing).
  - The inputi/inputs statement must update the specified variable to the value that was typed in by the user.
  - You do NOT need to handle the case where the user types an invalid input (e.g., the user types "abc" when inputi is called and expects an integer)

print statement

Used to print values out to the screen or pipe output into our automated testing framework.

- Syntax:

  ```
  (print arg1 arg2 ... argn)
  ```

  Where the arguments may be integer, string and boolean constants, variables referring to integer, string or boolean values, or expressions which evaluate to integer, string or boolean values. You will not be asked to print object references or null.

- Example usage:

  ```
  (class main
     (method main ()
        (print "here's a result " (* 3 5) " and here's a boolean" true)
     )
  )
  ```

  which prints:

  here's a result 15 and here's a booleantrue

- Requirements:
  - The print command must construct a string with the specified constant(s)/variable(s)/expression(s), with no added spacing or newlines and print it using our InterpreterBase.output() method.
  - Your implementation MUST use our InterpreterBase.output() method to actually output data to the user (so we can do automated testing).

return statement

Used to terminate the current method's execution and optionally return a value.

- Syntax - there are two variations:

  ```
  (return expression)  # terminates the current function
                       # and returns the value of the expression
  (return)             # terminates the current function, returns nothing
  ```

- Example usage:

```
(class main
   (method foo (q)
     (return (* 3 q)))   # returns the value of 3*q
   (method main ()
     (print (call me foo 5))
   )
)

(class main
   (method foo (q)
     (while (> q 0)
         (if (== (% q 3) 0)
           (return)  # immediately terminates loop and function foo
           (set q (- q 1))
         )
     )
   )
   (method main ()
     (print (call me foo 5))
   )
)
```

- Requirements:
  - The return statement must terminate the current method immediately, including breaking out of all (nested) while loops, begins, and if statements.
  - If the return statement does not have an argument, then the statement simply terminates the function but does not return a value to the calling function.
  - The returned value may be of any type

set statement

The set statement sets the value of a member variable (aka field) or a method's parameter to a specified new value.

- Syntax:

```
(set field_name value)
(set parameter_name value)
```

Where value can be a constant, a field name, a parameter name, or an expression.

- Example usage:

```
(class person
   (field name "")
   (field age 0)
   (method init (n a) (begin (set name n) (set age a)))
   (method talk (to_whom) (print name " says hello to " to_whom))
)

(class main
  (field x 0)
  (method foo (q)
    (begin
     (set x 10)               # setting field to integer constant
     (set q true)             # setting parameter to boolean constant
     (set x "foobar")         # setting field to a string constant
     (set x (* x 5))          # setting field to result of expression
     (set x (new person))     # setting field to refer to new object
     (set x null)             # setting field to null
    )
  )
  (method main ()
    (call me foo 5)
  )
)
```

- Requirements
  - The set statement changes the value of the specified field or parameter.
  - In Brewin v1 it is allowed to assign a variable a value of a different type than the variable referred to prior to the assignment.
  - We will not test assignment of the "me" object reference, so you may handle this (or not handle it) in any way you like:

    ```
    (set me something)   # we won't test this!
    ```

  - If a set statement attempts to assign a non-existent field or parameter to a value then you must generate an error of type ErrorType.NAME_ERROR by calling InterpreterBase.error().
  - We will not test you with a set statement that attempts to set a variable to the result of a function call (or any expression) that doesn't return a value. You can treat this as undefined behaviour.

while statement

Used to perform a while loop operation.

- Syntax:

```
(while expression statement_to_run_while_expression_is_true)
```

- Example usage:

```
(class main
  (field x 0)
  (method main ()
    (begin
     (inputi x)
     (while (> x 0)
       (begin
         (print "x is " x)
         (set x (- x 1))
       )
     )
    )
  )
)
```

- Requirements:
  - Make it work like a while loop in any language
  - The while loop's body must be a single statement. If you wish to have multiple statements run in the while loop, you must use a begin statement as the while loop's body
  - If the condition of the while loop does not evaluate to a boolean type, you must generate an error of type ErrorType.TYPE_ERROR by calling InterpreterBase.error().

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; for example, if and while statements is guaranteed to have at least a condition and a statement to execute if the condition is true
    - All variable names will start with a letter or underscore (not a number)
    - Note: if you do add checks for syntax errors, you're much more likely to save debugging time.
- WE WILL TEST YOUR INTERPRETER ON ONLY THOSE SEMANTIC and RUN-TIME ERRORS EXPLICITLY SPECIFIED IN THIS SPEC
    - You may 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 (adding a string and an int)
        - Using non-boolean expressions in if/while loop conditions
        - Passing an incorrect number of parameters to a method
        - Referring to a class name, method name, field or parameter that has not been defined
        - Dereferencing a null object reference, such as calling a method via an object reference field or parameter whose value is currently null.
    - 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 divide by zero, 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.

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. We will not test you on this, but it will be helpful for debugging. For example:

```
class Interpreter(InterpreterBase):
  …
  def interpret_statement(self, statement, line_num):
    print(f"{line_num}: {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 = ['(class main',
           '(method hello_world () (print “hello world!”))',
           ')']

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 output as we will not process this data in our testing framework.
- When you get input from the user (e.g., for an *inputi/inputs* statement) 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.
- To report errors (e.g., typing errors, name errors, run-time errors, and optionally syntax errors if you like) you must call the InterpreterBase.error() method with the specified error:

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

- We define all of the keywords for the Brewin language in our InterpreterBase class, e.g.:

```
# constants
CLASS_DEF = 'class'
METHOD_DEF = 'method'
FIELD_DEF = 'field'
NULL_DEF = 'null'
BEGIN_DEF = 'begin'
SET_DEF = 'set'
…
```

  You MUST use these constants in your interpreter source file rather than hard-coding these strings in your code.
- You must name your interpreter source file *interpreterv1.py*.

- You may submit as many other supporting Python modules as you like (e.g., *class.py*, *method.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 file since you will NOT be turning this file in. If your code depends upon a modified intbase.py file, 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., environment.py, type_module.py)

You MUST NOT submit intbase.py or bparser.py; we will provide our own. **You should 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 fifty 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.