Project #4: Brewin# Interpreter
CS131 Fall 2023
Due date: 12/03/23, 11:59pm

Warning: Expect project #4 to take 10-20 hours of time - it's about the same amount of work as project #3!

Warning: Do not clone our GitHub repo for project #3 if you use our solution, since this will force your cloned repo to be public. If/when folks cheat off your publicly-posted code, you'll have to explain to the Dean's office why you're not guilty of cheating.

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

Table of Contents

Introduction

The Brewin standards body (aka Carey) has met and has decided to add Object Oriented Programming features to the Brewin language, and will be updating the language's name to Brewin# in honor of these changes. In this project, you will be updating your Brewin interpreter so it supports these new Brewin# features. As before, you'll be implementing your interpreter in Python, and you may solve this problem using either your original project #3 solution, or by modifying the project #3 solution that we will provide.

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

What's New in Brewin#?

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

- Brewin# now has **support for objects** much like those used in JavaScript (Brewin# does NOT have classes, just objects). Objects may have member fields and methods
- Brewin# now supports **prototypal inheritance**, so an object can inherit methods/fields from a prototype object (and this can chain across multiple such prototype objects)
- Brewin# programs must also **support all operations on objects** (e.g., passing them as parameters, returning them, capturing them in closures, comparing them for equality, etc.)

Here are some examples:

Creating a new object and adding a field called x:

```
func main() {
  a = @;    /* @ is the symbol for creating a new object */
  a.x = 10; /* adds a field called "x" to the object, sets value to 10 */
  print(a.x);  /* prints 10 */
}
```

Adding a method to an object, calling the method, use of "this":

```
func main() {
  a = @;
  a.x = 10;
  a.member_func = lambda(p) {
    this.x = p;       /* sets a.x to value of p, using "this" keyword */
  };
```

```
  a.member_func(5);
  print(a.x);         /* prints 5 */
}
```

Support for prototypal inheritance:

```
func main() {
  /* define prototype object to represent an generic person */
  person = @;
  person.name = "anon";
  person.say_hi = lambda() { print(this.name," says hi!"); };

  carey = @;
  carey.proto = person;  /* assign person as carey's prototype object */

  carey.say_hi();        /* prints "anon says hi!" */
  carey.name = "Carey";
  carey.say_hi();        /* prints "Carey says hi!" */
}
```

What Do You Need To Do For project #4?

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

For this project, you will create a new class called *Interpreter* within a file called *interpreterv4.py* and derive it from our InterpreterBase class (found in our provided *intbase.py*). As with project #3, 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.

Objects

Instantiating a New Object

To define an empty new object (which has no fields or methods), use the following syntax:

```
variable_name = @;
```

You can think of this as:

obj_var_name = Object()

Adding Fields or Methods to an Object

Fields and methods are added to an object using the dot operator and assignment, as in Python:

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

func bar() {
  variable_name = @;

  /* set a field to a value */
  variable_name.field_name = value;

  /* add a method to the object which is a closure */
  variable_name.method1 = lambda(x) { print(x); };

  /* add a method to the object which is a function */
  variable_name.method2 = foo;
}
```

All fields and methods added to an object are public.

You must meet the following requirements when implementing adding fields/methods to an object:

- Assigning a new field/method within an object creates that field/method within the object
- Assigning an existing field/method to a new value/function changes the value of that field/method within the object

- Methods may be assigned to functions or to lambdas/closures
- You may assign a field to different types of values over time (e.g., { x = @; x.a = "hi"; x.a = 5; })
- Assignment of an object's field to a variable holding a primitive (e.g., integers, strings) results in the object's field pointing to a copy of the right-hand-side value, so changes to the field do not change the value referred to by the original right-hand-side variable, e.g.:

```
func main() {
  a = @;
  b = 5;
  a.x = b;
  a.x = 10;
  print(b); /* prints 5 */
}
```

- Assignment of an object field to a closure results in the object's method pointing directly at the original closure, NOT a shallow or deep copy of the closure, e.g.:

```
func main() {
  a = @;
  cap = 0;
  b = lambda() { cap = cap + 1; print(cap); };
  a.m = b;  /* points at same closure that b does */
  a.m();    /* prints 1 */
  a.m();    /* prints 2 */
  b();      /* prints 3, since a.m and b point to same closure  */
}
```

- Assignment of an object's field to another object q results in the object's field pointing directly at the original object q, NOT a shallow or deep copy of the original object, e.g.:

```
func main() {
  q = @;
  q.val = 10;

  b = @;
  b.field = q;   /* b.field points at original object, not a copy */

  tmp = b.field; /* tmp now points at same object as a does */
  tmp.val = 5;   /* setting tmp.val is really setting a.val */
```

```
  print(q.val);  /* prints 5 */
}
```

- You are NOT required to handle multiple-dot syntax, e.g., a.b.c = 5;
- You may assume that we will not use reserved keywords as fields in objects, and do not have to check for this (e.g., we will not do: x.while = 10;)
- It is acceptable for a field to have the same name as a variable or function elsewhere in the program, e.g.:

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

func bar() {
  x = @;
  x.foo = 5;   /* this is legal; the two foos are distinct */
}
```

Accessing Fields and Calling Methods in an Object

To access a method or field, simply use the dot operator:

```
print(variable_name.field_name);  /* prints value of field_name */

variable_name.method1(5);         /* calls method1, passing in 5 */
```

You must meet the following requirements when implementing access to fields and calls to methods of an object:

- You are NOT required to handle multiple-dot syntax, e.g., a.b.c = 5;
- Attempting to access a field that doesn't exist in an object must generate an error of ErrorType.NAME_ERROR.
- Attempting to call a method that doesn't exist in an object must generate an error of ErrorType.NAME_ERROR.
- Attempting to call a method in an object, e.g., x.foo(), where foo is not a function/lambda/closure (but perhaps an integer, string, boolean or another object) must generate an error of ErrorType.TYPE_ERROR.
- If an attempt is made to access a field or method on a non-object, this must result in an error of ErrorType.TYPE_ERROR, e.g.:

```
x = 5;
```

```
x.some_method();          /* x is not an object! TYPE_ERROR */
```

Method Execution

When a method executes, it operates identically to a regular function/closure with one exception - a new local variable, called "this", must be available in-scope during the method's execution. The "this" variable is an object reference which points to the object that was referred to on the left-hand-side of the dot during the method call. For example, in a call of q.foo(), the "this" object reference would refer to the same object referred to by variable q. Here's an example:

```
func foo() {
  print(this.x);      /* this refers to object q in the call to q.foo() */
  this.y = 20;
}

func main() {
  q = @;
  q.x = 10;
  q.foo = foo;        /* q.foo points at our foo function */

  q.foo();            /* prints 10, then sets q.y to 20 */
  print(q.y);         /* prints 20 */
}
```

You must meet the following requirements when supporting method execution:

- Methods must execute identically to traditional functions and closures with the exception that they now have the additional "this" variable in scope, which always refers to the object on the left-hand-side of the method call, e.g., c in c.foo().
- You may assume that a Brewin# programmer will never define a variable called "this" explicitly in their code. As such, your interpreter may have undefined behavior in this case, and may do anything it likes, including behaving differently than our canonical solution:
    - this = lambda() { print("hi"); };   /* illegal, so undefined behavior is OK */
- It is acceptable for a method to add/change methods within an object via the "this" object reference, e.g., this.foo = lambda(x) { this.var = x; };

Assignment of Objects

Assuming we have a variable x that refers to an object, and we assign a new variable y to x, both x and y will refer to the same object in memory (not a copy). For example:

```
func main() {
  x = @;
  y = x;           /* x and y refer to the same object in memory */
  y.field = 5;
  print(x.field);  /* prints 5 */
}
```

Passing Objects to Functions/Closures/Methods

Objects may be passed by value or by reference to a function/closure/method, just like any other type of variable.

Objects that are passed by value to a function will be deep-copied and have the deep copy passed to the function. Such a deep copy will also ensure that fields and methods (including any variables captured by those methods) are deep copied as well.

Modifications to objects that are passed by reference will affect the original passed-in argument.

Returning Objects from Functions/Closures/Methods

Objects are always returned by value (i.e., a deep copy is made of the returned object) just like with all other variables. Such a deep copy will also ensure that fields and methods (including any variables captured by those methods) are deep copied as well.

Comparing Objects

An object can be compared to another object or nil using the == and != operators. Two objects are considered equal if they have the same exact object reference (i.e., the two variables refer to the exact same object in memory). In all other cases, objects may not be considered equal - even, for example, if their values are the same.

```
func main() {
  x = @;
  x.a = 10;
  z = x;
```

```
  y = @;
  y.a = 10;

  if (x == x && x == z) { print("This will print out!"); }
  if (x == y) { print("This will not print!"); }
}
```

Variable Capture by Closures

As with project 3, all integers, booleans, and strings are still captured by value (via a deep-copy) in a closure. So changes to a captured primitive variable within a closure do not impact the value of that primitive outside the closure. For example:

```
func main() {
 c = 5;
 /* d captures object c by object reference */
 d = lambda() { c = 10; };

 d();
 print(c);  /* prints 5, since closure modified copy of variable c */
}
```

However, in Brewin# a variable referring to an object is captured in a closure *by reference*. In *capture by reference*, the captured variable inside the closure directly refers to the original variable defined outside the closure. Any changes to the captured variable within the closure impact the original variable outside the closure as well.

For example, in the following example, the line "c.x = 5;" impacts the original c object in function main() as opposed to a deep-copy held by closure d:

```
func main() {
 c = @;
 /* d captures object c by object reference */
 d = lambda() { c.x = 5; };

 d();
 print(c.x);  /* prints 5, since closure modified original object */
}
```

And, for example, in the following example, the line "c = @;" within the lambda impacts the original c object in function main() as opposed to a deep-copy held by closure d:

```
func main() {
 c = @;
 c.x = 5;

 /* d captures object c by object reference */
 d = lambda() {
   c = @;  /* changes original c variable, pointing it at a new obj */
   c.y = 10; /* adds field y to updated object */
 };

 d();
 print(c.y); /* prints 10 */
 print(c.x); /* NAME_ERROR since our original object is gone! */
}
```

In a similar fashion, in Brewin# all variables holding closures are captured reference. So if a closure d captures a variable c (that itself holds a closure), any changes to c within d will impact the outside variable c. For example:

```
func main() {
 c = lambda() { print(1); };

 /* d captures closure c by reference */
 d = lambda() { c = lambda() { print(2); }; };

 d();
 c();  /* prints 2, since c was captured by reference by d */
}
```

Prototypal Inheritance

As we discussed in class, prototypal inheritance is when a child object may specify another object as its prototype; the child object inherits all of the fields and methods of its prototype object, and this can extend multiple levels (c inherits from b, b inherits from a, etc.), e.g.:

```
func main() {
   p = @;
   p.x = 10;
   p.hello = lambda() { print("Hello world!"); };
```

```
    c = @;
    c.proto = p; /* c is a child of p, inherits x and hello() */
    c.y = 20;    /* c may have its own fields/methods too */

    c.hello(); /* prints "Hello world!" */
    print(c.x); /* prints 10 */
    print(c.y); /* prints 20 */
}
```

Your interpreter must support the requirements stated below to properly implement prototypal inheritance. For the requirements below, you may assume that we have a prototype object called p and a child object called c, e.g.:

- Assigning an object p as the prototype to object c is performed by writing:

  c.proto = p;

- A given object c may only have a single prototype object p at a time, though an object's prototype can be reassigned over time as needed.
- More than one object (e.g, c1, c2, c3) may specify the same object p as their prototype.
- Any field or method defined in a prototype object p will also be accessible in all of its children objects, unless (see next item)…
- If a field or method is defined in the child object c, the field/method in c will shadow any field/method of the same name in its prototype object p or any of p's prototype objects, e.g.:

```
func main() {
    p = @;
    p.x = 10;

    c = @;
    c.proto = p; /* c is a child of p, inherits x*/
    c.x = 20;    /* c.x now shadows p.x when we reference x thru c */

    print(c.x); /* prints 20 */
    print(p.x); /* still prints 10 */
}
```

- A method call to foo() in object c, e.g., c.foo(), will try to find a foo() method in object c. If foo() is found, the interpreter will run this version of the method. Otherwise, the interpreter will check object c's prototype object for a foo() method, and if one is present,

it will call that version. This process continues until the object no longer has a prototype to explore.

- When invoking a method foo() via a child object c, as in c.foo(), an object reference named "this" will be accessible within the scope of the foo() method as it executes. The object reference, named "this", will always point at object c (the object referenced in the initial call, **c**.foo()), irrespective of whether the executing foo() method is defined within the child object c or one of its prototype objects (e.g., p1, p2, …). From this, we get the following:

    - Any new variables/methods added via syntax like:

        this.field = value;
        this.method = lambda() { … };

      by an executing method, regardless of whether that executing method is defined in a prototype object or in the child object, will always be added to the child object, since "this" always refers to the child object:

```
func main() {
  /* define prototype object */
  p = @;
  p.add_foo = lambda() { this.foo = 10; };

  /* define child object c, which inherits from p */
  c = @;
  c.proto = p;
  c.add_foo();

  print(c.foo);  /* prints 10 */
  print(p.foo); /* NAME_ERROR since foo not defined in p */
}
```

    - Assume method m() is defined in object c or one of its prototypes, and we call it via the child object, c.m(), and further that method m() invokes another method q(), as in this.q(). Upon the call to this.q(), the interpreter initiates a lookup for q(), beginning with the object referenced by the "this" keyword (which is *always* the child object c). If q() is not found there, the search continues up the prototype chain, checking each prototype object in succession for q(). This continues until the q() method is either located or it becomes evident that no such method exists in the object's prototype hierarchy. For example, the following program will print "Carey says hello!":

```
func main() {
  person = @;
  person.name = "anon";
  person.act = lambda() { this.say_hi(); };
  person.say_hi = lambda() { print(this.name," says hi!"); };

  carey = @;
  carey.name = "Carey";
  carey.proto = person;
  carey.say_hi = lambda() { print(this.name," says hello!"); };

  carey.act(); /* prints "Carey says hello!" */
}
```

Why? When we call carey.act(), the "this" object reference in the person.act() method will refer to the carey object (even though the act() method is defined in the person object). So the call to this.say_hi() must call carey.say_hi() and not person.say_hi().

- Assume we have an object c, which has a prototype object of p1, which itself has a prototype object of p2, and so on up to object p10 (c -> p1 -> p2 -> p3 -> … -> p10). Further assume that object p5 defines a method foo(), but none of the other objects define a method named foo(). If we call c.foo(), the foo() method in p5 will run. It may use every method/field defined in c or any of its prototypes (p1-p10), regardless of whether they're defined above or below p5[1]. For example:

```
func main() {
  p = @;
  p.foo = lambda() {
    this.bar(5);  /* calls bar() even though bar() not defined in p */
    print(this.x); /* prints 10, even tho x not defined in p */
  };

  c = @;
  c.proto = p;
  c.bar = lambda(x) { print(x); };
  c.x = 10;

  c.foo();  /* the call to bar() works! this prints 5, then 10 */
  p.bar();  /* NAME_ERROR - bar is not known to object p */
```

[1] Of course, if a field f is defined in p10 and also in c, then the version defined in c will shadow the version defined in p10, and our method in p5 will therefore use the value present in c.

```
}
```

- Changes may be made to a prototype object after one or more child objects have been assigned to refer to the prototype object, and those changes must subsequently be reflected in all children object(s), e.g.:

```
func main() {
  p = @;

  c = @;
  c.proto = p;

  p.x = 10;    /* change proto object after c refers to p */
  print(c.x);  /* change is visible in c, this prints 10 */
}
```

- The programmer may also refer to the proto field of an object in the right-hand-side of an expression, e.g.: my_proto = obj.proto; If the field has not been defined, reference to it must result in an ErrorType.NAME_ERROR.
- Cyclical linking of protos is not allowed (e.g., a.proto = b; b.proto = a;). In such a case, your interpreter may have undefined behavior in this case, including behaving differently than our canonical solution.
- You may assign a proto field to nil. Doing so is equivalent to an object not having a proto field at all.
- There is NO notion of a "super" object reference in Brewin#, so if a prototype object p defines a function foo() and child object c also defines foo(), there is no way for the child object to call p's version of foo().
- You are NOT required to handle overloaded methods across objects. If the Brewin# programmer defines method foo(a) in the prototype object and foo(a,b) in the child object, your interpreter may do anything it likes, including behaving differently than our canonical implementation.
- If you specify a non-object (e.g., an int, bool or string) as a prototype object, then this must generate an error of ErrorType.TYPE_ERROR.
- Access to a field, e.g., name, that is not defined in c or its prototype p, or p's prototype, etc, must generate an error of ErrorType.NAME_ERROR.
- A method call to a method, e.g., bar(), that is not defined in c or its prototype p, or p's prototype, etc, must generate an error of ErrorType.NAME_ERROR.

Abstract Syntax Tree Spec

As in project #3, the AST will contain a bunch of nodes, represented as Python *Element* objects. You can find the definition of our *Element* class in our provided file, *element.py*. Each *Element* object contains a field called *elem_type* which indicates what type of node this is (e.g., function, lambda, statement, expression, variable, reference argument, …). Each *Element* object also holds a Python dictionary, held in a field called *dict*, that contains relevant information about the node.

Here are the different types of nodes you must handle in project #4, with changes from project #3 **bolded for clarity**:

Program Node

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

A Program Node will have the following fields:

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

Function Definition Node

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

A *Function Node* will have the following fields:

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

Lambda Definition Node

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

A *Lambda Node* will have the following fields:

- self.elem_type whose value is 'lambda', identifying this node is a *Lambda Node*
- self.dict which holds two keys
    - 'args' which maps to a list of *Argument* or *Reference Argument* nodes

    - 'statements' which maps to a list of Statement nodes, representing each of the statements that make up the function, in their order of execution

Argument Node

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

An *Argument Node* will have the following fields:

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

Reference Argument Node

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

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

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

Statement Node

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

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

- self.elem_type whose value is '='
- self.dict which holds two keys
    - 'name' which maps to a string holding the name of the variable on the left-hand side of the assignment (e.g., the string 'bar' for bar = 10 + 5;)**; the left-hand side variable may use dot notation so you may now refer to object fields (e.g., x.bar = 10 + 5;); only a single dot is allowed (e.g., a.b.c = 5; is not allowed, nor does your interpreter need to check for this case)**

    - '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 a method call will have the following fields:

- **self.elem_type whose value is 'mcall'**
- **self.dict which holds three keys**
    - **'objref' which maps to the variable name on the left-hand-side of the dot. For example, in bar.foo(), objref would map to the string 'bar'. This field could hold a variable name or the "this" keyword.**
    - **'name' which maps to the name of the function that is to be called in this statement (e.g., the string 'print', or 'foo' as in bar.foo())**
    - **'args' which maps to a list containing zero or more *Expression* nodes, *Variable* nodes or *Value* nodes that represent arguments to the function call**

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

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

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

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

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

- self.elem_type whose value is 'return'

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

Expression Node

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

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

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

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

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

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

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

An *Expression* node representing a method call (e.g. bar.foo(7)) will have the following fields:

- **self.elem_type whose value is 'mcall'**
- **self.dict which holds three keys**
    - **'objref' which maps to the variable name on the left-hand-side of the dot. For example, in bar.foo(), objref would map to the string 'bar'. You may also use the "this" keyword for your objref**

    - **'name' which maps to the name of the function being called, e.g. 'factorial', or in the case of bar.foo() it would be 'foo'**
    - **'args' which maps to a list containing zero or more *Expression* nodes, *Variable* nodes or *Value* nodes that represent arguments to the function call**

An *Expression* node representing instantiation of a new object will have a single field:

- **self.elem_type whose value is '@'**
- **There are no other fields/dictionaries in this node**

Variable Node

A *Variable* node represents an individual variable that's referred to in an expression or statement. **It may also refer to a field or method inside an object (e.g., x.field_name, x.method_name)**:

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' **or 'x.y' to access field y inside object 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, #2 and #3, and:

- You must name your interpreter source file *interpreterv4.py*.
- You may submit as many other supporting Python modules as you like (e.g., *statement.py*, *variable.py*, …) which are used by your *interpreterv4.py* file.
- You MUST NOT modify our intbase.py, brewlex.py, or brewparse.py files since you will NOT be turning these file in. If your code depends upon modified versions of these files, this will result in a grade of zero on this project.

Deliverables

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

- Your interpreterv4.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 interpreterv4.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.