Project #3: Brewin++ Interpreter
CS131 Fall 2024
Due date: 11/17 11:59pm (Sunday)

Important Info:

- Expect project #3 to take 20-30 hours of time. **Start early!!!**

- If you use our solution, ***do not clone our GitHub repo for project #2***, since it 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. Instead, copy-paste the solution files into your existing environment!

- Make sure to initialize a new git repo before starting! Just like project #2, we *might* ask you to submit your git history! More info about this here

- Your gradescope score is your **final score** and you may submit to gradescope as many times as you'd like

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

Table of Contents

Introduction

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

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

So what new language features were added to Brewin++? Here's the list:

Static Typing

should have a something like type_value maybe but for variables

Brewin++ now implements static typing[1], meaning all variables and parameters have fixed types, and functions have explicit return types. Your program must now check that all types are compatible (e.g., a string variable can't be assigned to or compared with an int value, a string can't be passed to a Boolean parameter, a function with a bool return type can't return a string value, etc).

Here's an example program which shows types added for variables and parameters, and for function return types:

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

class Variable:
# represents a variable which has a name, type, & value

```
func main() : void {
  var n : int;
  n = inputi("Enter a number: ");
  print(fact(n));
}

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

[1] Even though Brewin++ is an interpreted language, by adding variable definitions and functions with types, we enable all variable's types to be determined prior to execution, so a compiler could be written if we desired, making this a statically-typed language. But technically, it's still dynamically typed for now - that is, type checking is performed dynamically at runtime.

The above program would print:

Enter a number:
5
120

Default Return Values from Functions

In Brewin++, if a function has a non-void return type then it always returns a value. If the statements within the function's body do not explicitly return a value using a return statement (e.g., return 5;, then the interpreter must return the default value for the function's return type upon the function's completion (e.g., 0 for ints, false for Booleans, "" for Strings, etc). This way, all non-void functions return some value even if they don't explicitly have a return statement to do so. So, for example:

```
func main() : void {
  print(foo());
  print(bar());
}

func foo() : int {
  return; /* returns 0 */
}

func bar() : bool {
  print("bar");
}  /* returns false*/
```

```
The above program would print:
0
bar
false
```

Coercion

Brewin++ now supports limited coercion from integers to boolean values. No other types of coercion (e.g., from boolean to integer, from string to integer, etc.) are supported in Brewin++. Here is a non-exhaustive list of examples:

```
func main() : void {
  print(5 || false);
  var a:int;
  a = 1;
  if (a) {
    print("if works on integers now!");
  }
  foo(a-1);
}

func foo(b : bool) : void {
  print(b);
}
```

The above program would print:

true
if works on integers now!
false

Structures

Brewin++ now supports user-defined structures like those in C++. To allocate a structure, you must use the "new" command. All struct variables (like p below) are object references. For example:

```
struct Person {
  name: string;
  age: int;
  student: bool;
}

func main() : void {
  var p: Person;
  p = new Person;
  p.name = "Carey";
  p.age = 21;
  p.student = false;
  foo(p);
}
```

```
func foo(p : Person) : void {
  print(p.name, " is ", p.age, " years old.");
}
```

The above program would print:

Carey is 21 years old.

What Do You Need To Do For Project #3?

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

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

Brewin Language Spec

The following sections provide detailed requirements for the Brewin++ language so you can implement your interpreter correctly. All language features not covered in these sections must function exactly as they did in the original Brewin language.

Function Definitions

All Brewin functions now must have explicit types for their parameters and return value.

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

```
func function_name(arg1 : type1, arg2: type2, ...) : return_type {
    statement_1;
    statement_2;
    ...
    statement_n;
}
```

Here is an example showing how to define several valid Brewin functions:

```
func foo(a:int, b:string, c:int, d:bool) : int {
  print(b, d);
  return a + c;
}

func talk_to(name:string): void {
  if (name == "Carey") {
     print("Go away!");
     return;  /* using return is OK w/void, just don't specify a value */
  }
  print("Greetings");
}

func main() : void {
  print(foo(10, "blah", 20, false));
  talk_to("Bonnie");
}
```

Every Brewin function must now have a type for every parameter. Valid types that may be used for variables include:

- int
- string
- bool
- user-defined struct types (see the struct section below)

Overloading by the number of parameters must still be supported, but overloading by type with the same number of parameters is not supported and will not be tested. For example, we will not test for these types of cases:

```
func foo(a:int) : int {
  return a+1;
}

func foo(a:string) : string {
  return a + " hi";
}

func main(): void {
   foo("hi");  /* not supported */
```

maybe also have a class to represent fxns
↳ members: parameter list (stores variable objects), return type

```
}
```

Every Brewin function must now have a return type. For a function's return type, you may use any of the above types as well as the "void" return type which indicates that the function returns no value. Void functions may use a "return;" statement to terminate their execution, but may never return a value, as in "return 5;".

Use of an invalid/undefined/missing type for a parameter or return type must result in an error of ErrorType.TYPE_ERROR. This check should happen before the execution of the main function, and the error should be reported no matter if the function in question is used in execution or not.

will need to check ast first for this? ↳ how best to approach?

Variable Definitions

All variable definitions must now specify an explicit type for the variable, e.g.:

```
var eyeballs: int;
var theory: string;
var old: bool;
var n: node;  /* assuming we have defined a node structure - see below */
```

All variables must be initialized with the default value for their type:

- bool: false
- int: 0
- string: ""
- user-defined structures: nil

→ in variable class, have a get_default fxn or smth. or store defaults as dict.

You may not specify an initial value for a variable at definition time. This is not supported in Brewin++ (i.e., we will not test it):

```
var name: string = "Carey"; /* NOT SUPPORTED!!! */
```

If a type used to define a variable is invalid/unknown, then you must generate an error of type ErrorType.TYPE_ERROR. This error is reported only when the variable definition statement is executed.

User-defined Structures

The programmer may define 0 or more user-defined structures in their Brewin++ program. All such structures must be defined at the top of the program, before the definition of the first function. The syntax for defining a structure is as follows:

```
struct StructTypeName {
   field1: type1;
   field2: type2;
   ...
   fieldn: typen;
}
```

Where the field types may be any primitive type or a struct type defined earlier in the source file. For example:

```
struct cat {
  name: string;
  scratches: bool;
}

struct person {
  name: string;
  age: int;
  address: string;
  kitty: cat;
}
```

Note that self-referential structures are allowed so you can create structures like linked lists:

```
struct node {
  value: int;
  next: node;    /* this works since node is defined above! */
}
```

But, for example, this would not be allowed:

```
struct person {
  name: string;
  age: int;
  address: string;
  pupper: dog;      /* not allowed - dog is defined below person!!! */
}
```

```
struct dog {
  name: string;
  vaccinated: bool;
}
```

References to undefined or not-yet-defined structs in other structs is considered undefined behavior and will not be tested.

Defining a variable or field of a struct type results in the definition of an object reference (e.g., a pointer) with a value of nil. It does not result in allocation of a full struct instance with fields. So for example:

```
struct dog {
  name: string;
  vaccinated: bool;
}

func main() : void {
  var d: dog;    /* d is an object reference whose value is nil */

  print (d == nil);  /* prints true, because d was initialized to nil */
}
```

To allocate a struct you must use the *new expression*, which has been introduced in Brewin++. The new expression has the following syntax:

new struct_type

have to check for this?

```
struct dog {

}


func main() : void {
  var my_dog : dog;
  print(my_dog);
}
```

as in:

kippy = new dog;

or:

f(new dog); / allocates new dog and passes its object reference to function f */*

When you use the new expression, it allocates a new structure, initializes each of the structure's fields to its default value based on the field's type, and returns an object reference to the new structure. If a type used with a new expression is not a valid struct type (defined in the structs section of the program), then you must generate an error of type ErrorType.TYPE_ERROR.

For example:

Note on functionality

```
struct dog {
  name : string;
}

func main() : void {
  var my_dog : dog;
  print(my_dog);
  my_dog = new dog;
  print(my_dog);
  print(my_dog.name);
    my_dog.name = "max";
  /*print(my_dog);*/
    print(my_dog.name);
}
```

my_dog is now nil
prints nil
now my_dog looks like: my_dog {name: ""}
THROWS ERROR
prints ""
now: my_dog {name: "max"}
prints "max"

```
struct flea {
  age: int;
  infected : bool;
}

struct dog {
  name: string;
  vaccinated: bool;
  companion: flea;
}

func main() : void {
  var d: dog;
  d = new dog;    /* sets d object reference to point to a dog structure */

  print(d.vaccinated); /* prints false - default bool value */
  print(d.companion); /* prints nil - default struct object reference */

  /* we may now set d's fields */
  d.name = "Koda";
  d.vaccinated = true;
  d.companion = new flea;
  d.companion.age = 3;
}
```

note if do this @ end here, throws error

The above program would print:

false
nil

As shown above, to access field values in a struct, you may use the dot operator. The dot operator may only be used with local variables, parameters and fields (e.g., person.name = "lanny", not values, foo().bar = 5;). As shown above, more than one dot operator may be used in a single expression to access nested fields.

When implementing field access, you must do the following:

- If, during execution, the variable to the left of a dot is not a struct type, then you must generate an error of ErrorType.TYPE_ERROR.
- If, during execution, the variable to the left of a dot is nil, then you must generate an error of ErrorType.FAULT_ERROR.

- If, during execution, a field name is invalid (e.g., it's not a valid field in a struct definition), then you must generate an error of ErrorType.NAME_ERROR.

Parameter Passing

Parameter passing in Brewin++ works as follows:

fine bc in python they are immutable

- All primitives (int, bool, string) are passed by value and their value may not be changed by a function to which they are passed
- All struct variables/values are passed by object reference, and thus the contents of a passed-in struct may be mutated by a called function.
- If an object reference to a struct is passed to a function, assignment of that object reference within the called function will NOT affect the caller's object reference variable

So for example:

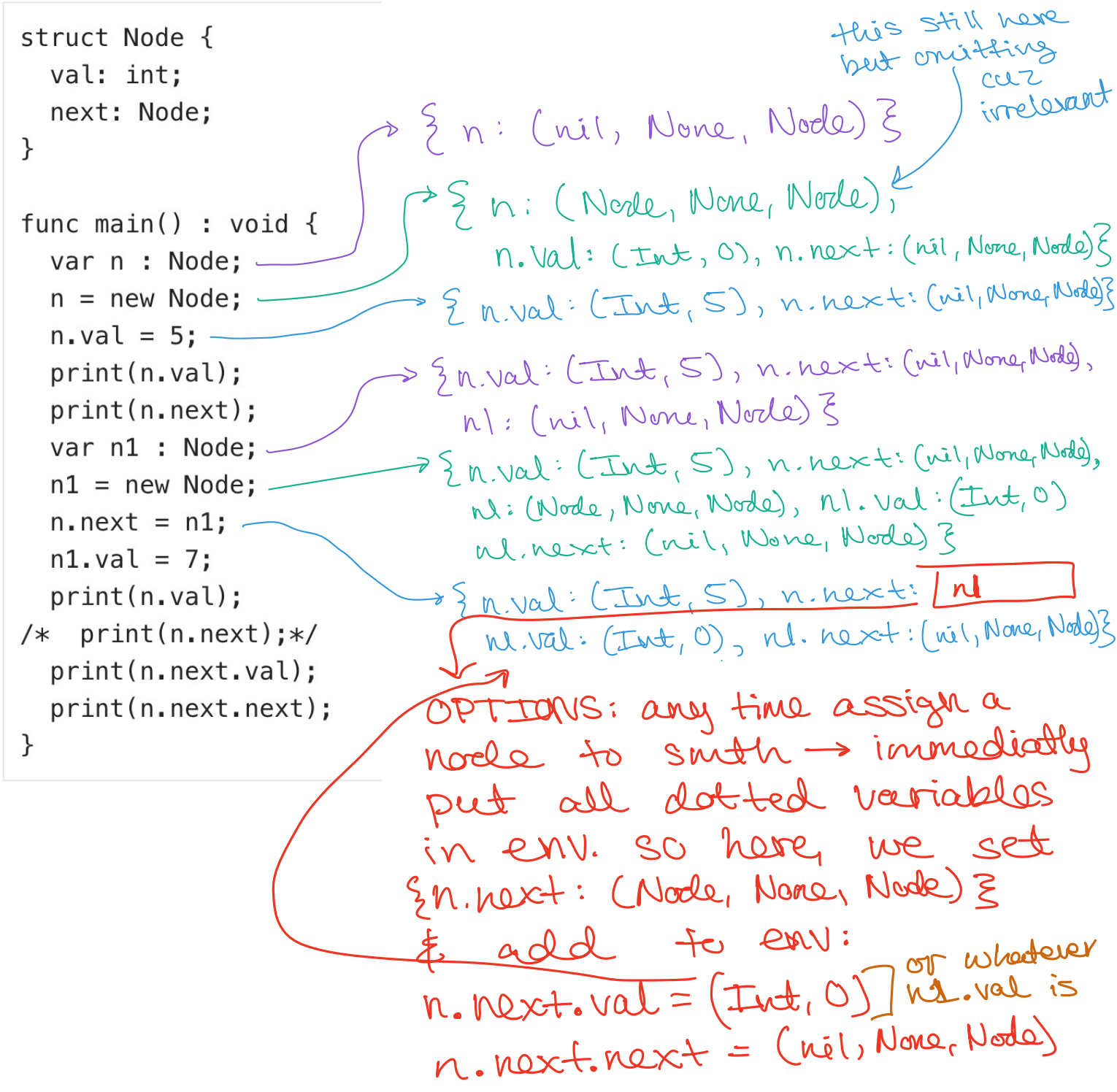
```
struct person {
  name: string;
  age: int;
}

func foo(a:int, b: person) : void {
  a = 10;
  b.age = b.age + 1;  /* changes p.age from 18 to 19 */

  b = new person;  /* this changes local b variable, not p var below */
  b.age = 100;     /* this does NOT change the p.age field below */
}

func main() : void {
  var x: int;
  x = 5;
  var p:person;
  p = new person;
  p.age = 18;
  foo(x, p);
  print(x);      /* prints 5, since x is passed by value */
  print(p.age);  /* prints 19, since p is passed by object reference */
}
```

The above program prints:


struct Node {
val: int;
next: Node;
}
func main() : void {
var n : Node;
n = new Node;
n.val = 5;
print(n.val);
print(n.next);
var n1 : Node;
n1 = new Node;
n.next = n1;
n1.val = 7;
print(n.val);
/* print(n.next);*/
print(n.next.val);
print(n.next.next);
}
{ n: (nil, None, Node) }
this still here but omitting cuz irrelevant
{ n: (Node, None, Node),
n.val: (Int, 0), n.next: (nil, None, Node)}
{ n.val: (Int, 5), n.next: (nil, None, Node)}
{n.val: (Int, 5), n.next: (nil, None, Node),
n1: (nil, None, Node) }
{n.val: (Int, 5), n.next: (nil, None, Node),
n1: (Node, None, Node), n1.val: (Int, 0)
n1.next: (nil, None, Node) }
{ n.val: (Int, 5), n.next: n1
n1.val: (Int, 0), n1.next: (nil, None, Node)}
OPTIONS: any time assign a node to smth → immediately put all dotted variables in env. so here, we set
{n.next: (Node, None, Node) }
& add to env:
n.next.val = (Int, 0)
or whatever n1.val is
n.next.next = (nil, None, Node)

check nesting smth not defined

A passed in variable/value must have a compatible type with the type of the formal parameter it is passed to. Specifically:

- A variable/value of primitive type T (e.g., int, bool, string) may be passed to a formal parameter that is of type T
- A variable/value of type struct of type T may be passed to a formal parameter of type T
- A variable/value of type int may be passed to a formal parameter that is of type bool, via a coercion (see below)
- A value of nil may be passed to a formal parameter of a structure type T (but ***not*** to a formal parameter of ***primitive*** type T)

If a program passes an argument of an invalid type to a function's formal parameter (e.g., passing a bool to a function that accepts an int parameter) then your program must generate an error of ErrorType.TYPE_ERROR.

The return statement

Return statements in Brewin++ work just like return statements in Brewin, but must support type checking. If a return statement returns a value, then the type of that value must match the function's return type or be compatible with the function's return type via a coercion. Here are the rules:

- A variable/value of primitive type T (e.g., int, string, bool) may always be returned by a function with a return type of T
- A variable/value of a user-defined structure type T may always be returned by a function with a return type of T
- A variable/value of type int may be returned by a function with a return type of bool, via a coercion (see below)
- A value of nil may be returned by a function with a return type of a user-defined structure T (but ***not*** a function with a ***primitive*** return type)

If the type of a returned value is incompatible with the function's return type (e.g., returning an int when the function's return type is bool), you must generate an error of ErrorType.TYPE_ERROR.

If a function either uses "return;" and does not specify a value, or the function does not explicitly use a return statement and runs to completion, then the function must return the default value for the function's return type. Here's a list function function return types and the default values that must be returned for each type:

- bool: false
- int: 0
- string: ""
- user-defined structures: nil

For example:

```
struct dog {
  bark: int;
  bite: int;
}

func bar() : int {
  return;  /* no return value specified - returns 0 */
}

func bletch() : bool {
  print("hi");
  /* no explicit return; bletch must return default bool of false */
}

func boing() : dog {
  return;  /* returns nil */
}

func main() : void {
   var val: int;
   val = bar();
   print(val);  /* prints 0 */
   print(bletch()); /* prints false */
   print(boing()); /* prints nil */
}
```

This program prints:

0
hi
false
nil

A void function may use a return statement but must do so without specifying a return value (e.g., "return;"). If a void function tries to return any type of value (e.g., return 5;), then this must generate an error of ErrorType.TYPE_ERROR.

Invoking a void return type function as part of an expression should always throw an error of ErrorType.TYPE_ERROR.

In a function with a primitive return type (e.g., int, bool, string), the return statements will return by value. In a function with a user-defined struct return type, return statements return the object reference of any struct.

Here's an example of passing and returning a user-defined struct.

same as python p much

```
struct dog {
 bark: int;
 bite: int;
}

func foo(d: dog) : dog {  /* d holds the same object reference that the
koda variable holds */
  d.bark = 10;
  return d;             /* this returns the same object reference that the
koda variable holds */
}

 func main() : void {
  var koda: dog;
  var kippy: dog;
  koda = new dog;
  kippy = foo(koda);    /* kippy holds the same object reference as koda */
  kippy.bite = 20;
  print(koda.bark, " ", koda.bite); /* prints 10 20 */
}
```

The above program prints:

10 20

Assignments

Assignments work just as they did in Brewin, but must support type checking.

- A variable of primitive type T (e.g., int, string, bool) may always be assigned by using a value or variable of type T

n1.next = n2.next.next.val

{val: ( )
next: (Node, fields{ }}

h2 : {val: ( ), fields

- A variable of a user-defined structure type T (i.e., an object reference of type T) may always be assigned by using the object reference of another value or variable with the same type T
- A variable of a user-defined structure type T (i.e., an object reference of type T) may always be assigned by using a value of type nil. No other variable of any type may be assigned to nil.
- A variable of type bool must also support assignment by using a value or variable of type int, via a coercion (see below)
- No other type combinations for assignments are valid

If the types of the target variable and source value are incompatible, you must generate an error of ErrorType.TYPE_ERROR.

Coercions

Brewin++ supports limited coercions from integer values/variables to boolean values/variables. No coercions between other types are supported (e.g., from string to int, or bool to int). The int → bool coercion implicitly converts an integer value into a corresponding boolean value, using the following rules:

- if the integer is zero, then it is coerced into a value of false
- if the integer is non-zero (including negative values), then it is coerced into a value of true

Coercions must be supported in all of the following places:

- Assignment: assigning an integer value/variable to a boolean variable
- Parameter passing: passing an integer value/variable to a function that has a boolean formal parameter
- Returning values: returning an integer value/variable from a function that has a boolean return type
- If statements: using an integer value/variable as the condition for an if statement: if (some_int_variable) { /* do this */ }
- For statements: using an integer value/variable as the condition for a for statement e.g., for (k = 5; k ; k = k - 1)
- && and || expressions, e.g., checking the value of an integer in an and/or expression, e.g., if (int_variable || bool_variable && other_int_variable) { /* do this */ }
- == and != expressions, e.g., 5 == true would be true, false == 0 would be true

Coercions are not supported under any other circumstances (e.g., false < 5); all must result in errors of type ErrorType.TYPE_ERROR.

Comparisons

Comparisons work just as they did in Brewin, but must support type checking (and coercion, as described above).

- A variable of primitive type X (e.g., int, string, bool) may always be compared to a value or variable of type X
- A variable of a user-defined structure type T (i.e., an object reference of type T) may always be compared to a value or variable with the same type T
  - Struct variables are compared based on their object reference (e.g., two variables are only equal if they point to the exact same object, NOT if two different objects are pointed to, even if their field values are identical)
- A variable of type bool may always be compared to a value or variable of type int, via a coercion (see below), e.g., all of the following are true statements:
  - 5 == true, true == 1, -5 == true, 0 == false, true != 0, etc.
- A variable of a user-defined structure type T (i.e., an object reference of type T) may always be compared to a value of type nil
- No other types of values other than struct values/variables may be compared to nil

use is to check equality here

If the types of the compared values are incompatible, you must generate an error of ErrorType.TYPE_ERROR. Similarly, an attempt to compare a void type (e.g., the return of print()) to any other type must result in an error of ErrorType.TYPE_ERROR.

You will never be asked to compare two struct types that are currently nil, and may have undefined behavior in this case.

Other Changes

Print Function

The print() function still may accept any number of parameters of any type. However, its return type is now void, and it no longer returns nil. Therefore you must not assign anything to print's return value, nor use it in expressions.

The print() function must now be able to print out nil values, e.g., print(nil); must work and print out "nil". Similarly, you must be able to print a nil reference to an object as in the following example:

```
struct dog {
  name: string;
```

```
  vaccinated: bool;
}

func main() : void {
  var d: dog;    /* d is an object reference whose value is nil */

  print (d);  /* prints nil, because d was initialized to nil */
}
```

You will never be asked to print out a non-nil object reference, and may have undefined behavior in this case.

Abstract Syntax Tree Spec

As in project #2, the AST will contain a bunch of nodes, represented as Python *Element* objects. You can find the definition of our *Element* class in our provided file, *element.py*. Each *Element* object contains a field called *elem_type* which indicates what type of node this is (e.g., function, 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 #3, with changes from project #2 **bolded** for clarity:

Program Node

A *Program* node represents the overall program, and it contains **a list of Struct nodes that define all of the user-defined structures** and *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 two keys:**
  - **'structs' which maps to a possibly-empty list of Structure Definition Nodes representing each of the structures defined in the program**
  - 'functions' which maps to a list of *Function Definition* nodes representing each of the functions in the program

maybe add this as member to environment

Struct Definition Node

A *Struct Definition* Node represents an individual structure definition, and it contains the structure's name (e.g. 'node'), and the list of fields that are part of the structure:

A *Struct Node* will have the following fields:

- **self.elem_type whose value is 'struct', identifying this node is a *Structure* node**
- **self.dict which holds two keys**
  - **'name' which maps to a string containing the name of the structure (e.g., 'node')**
  - **'fields' which maps to a list of the structure's fields; each field's definition is held in a Field Definition node**

Field Definition Node

A *Field Definition* node represents the definition of an individual field within a struct. It contains the field's name (e.g. 'x'), and its type (e.g., 'int' or 'node').

A *Field Definition Node* will have the following fields:

- **self.elem_type whose value is 'fielddef', identifying this node is a *Field Definition* Node which defines a new field**
- **self.dict which holds two keys**
  - **'name' which maps to a string containing the name of the field (e.g., 'age')**
  - **'var_type' which is the type of this field (e.g., int, bool, node, etc.)**

Function Definition Node

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

A *Function Node* will have the following fields:

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

n2.next=nil
n2
n1

n.next = n2 ⟶ n.next.val = n2.val
n.next.next = n2.next
n.next.next = n2.next.next

If set to nil, need to clear

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 two keys**
    - 'name' which maps to a string holding the name of the formal parameter, e.g. 'x' in func f(x:int) { … }
    - **'var_type' which maps to a string holding the type of the formal parameter, e.g., 'int' in func f(x:int) : void { … }**

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 #3, this will be one of the following:

A *Statement* node representing a variable definition will have the following fields:

- self.elem_type whose value is 'vardef'
- **self.dict which holds two keys**
    - 'name' which maps to a string holding the name of the variable to be defined
    - **'var_type' which is the type of this variable/field (e.g., int, bool, node, etc.)**

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

- self.elem_type whose value is '='
- self.dict which holds two keys
    - 'name' which maps to a string holding the name of the variable on the left-hand side of the assignment (e.g., the string 'bar' for bar = 10 + 5;)
    - 'expression' which maps to either an *Expression* node (e.g., for bar = 10+5;), a *Variable* node (e.g., for bar = bletch;) or a *Value* node (for bar = 5; or bar = "hello";)

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

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

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

- self.elem_type whose value is 'if'
- self.dict which holds three keys

    - 'condition' which maps to a boolean expression, variable or constant that must be True for the if statement to be executed, e.g. x > 5 in if (x > 5) { … }
    - 'statements' which maps to a list containing one or more statement nodes which must be executed if the condition is true
    - 'else_statements' which maps to None (when the if-statement doesn't have else clause) or a list containing one or more statement nodes which must be executed if the condition is false

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

- self.elem_type whose value is 'for'
- self.dict which holds four keys
    - 'init' which maps to an assignment statement (e.g., x = 0)
    - 'condition' which maps to a boolean expression, variable or constant that must be true for the body of the for to be executed, e.g. x > 5 in for (x=0; x > 5; x = x + 1) { … }
    - 'update' which maps to an assignment statement that perform updating of the looping variable (e.g., x = x + 1)
    - 'statements' which maps to a list containing one or more statement nodes which must be executed if the condition is true

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

- self.elem_type whose value is 'return'
- self.dict which holds one key
    - 'expression' which maps to an expression, variable or constant to return (e.g., 5+x in return 5+x;), or None (if the return statement returns a default value of nil)

Expression Node

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

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

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

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

- self.elem_type whose value is either 'neg' for arithmetic negation or '!' for boolean negation

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

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

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

An *Expression* node representing a new command (e.g. "new dog" in "d = new dog;") will have the following fields:

- **self.elem_type whose value is 'new'**
- **self.dict which holds one key**
    - **'var_type which maps to the type name of the structure to allocate**

Variable Node

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

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 four types of *Value* nodes (representing integers, strings, booleans and nil 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")

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

- self.elem_type whose value is 'bool'

WONT be tested on this

- 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:

```
func main() {
    print("hello world");
}struct Person {
  name: string;
  age: int;
  student: bool;
}

struct dog {
  name: string;
  parent: Person;
}

struct node {
  val: int;
  next: node;
}

/*NOT TESTED ON THIS*/
func main() : void {
  var p: Person;
  print(p);
  print("this prints?");
  var d: dog;
  if (p == d) {
    print("Does this print");
  }
  print("end");
}
```

your stdin

- 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 not create types, variables, parameters or field names using reserved words in our tests
- 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 function
        - 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.

Representing Structs

[[{}, {'num': (Int, 0)}
{'my_dog': (Dog, {})}]]

struct:
  type
  fields

] before returning anything, check if type in struct_list or not

except in case of self-referential!

* Remember, every struct contains primitives @ their lowest level (ex if a struct is a field of another struct, eventually it leads to primitives or self referential).

Extend value:

when we new a value, set value to "ERROR"

Value():
  type:
  value:
  fields: ← None for all none-structs & defined but not-newed structs

Coding Requirements

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

- You must name your interpreter source file *interpreterv3.py*.
- You may submit as many other supporting Python modules as you like (e.g., *statement.py*, *variable.py*, …) which are used by your *interpreterv3.py* file.
- Try to write self-documenting code with descriptive function and variable names and use idiomatic Python code.
- You MUST NOT modify our intbase.py, brewlex.py, or brewparse.py files since you will NOT be turning these files in. If your code depends upon modified versions of these files, this will result in a grade of zero on this project.
- We **strongly** encourage you to keep an incremental git history documenting your progress on the project — though it is not required. In case your code resembles another submission, this is a great way to prove you did the work yourself!
    - In general, it's a good idea to commit when you add a feature, fix a bug, or refactor some code. Here's an example of an ideal commit history. Here's an example of a commit history that is lacking.

Deliverables

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

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

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

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

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

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

Grading

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

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

We strongly encourage you to write your own test cases. The TAs have developed a tool called barista 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. In other words, the grade you see on GradeScope is your final grade![2]

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 our 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)

[2] As long as you submit on time.

    - 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
    - Any attempts to cause our framework to give you a different score/grade then your solution would otherwise earn
- 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 so long as you include a citation in your comments. See the syllabus for more concrete guidelines surrounding LLMS.
- Using ChatGPT, CoPilot or similar code generation tools to generate any number of test cases, test code, etc.