Project #3: Brewin#
CS131 Fall 2022
Due date: Nov 22nd, 11:59pm

Introduction

The Brewin standards body has met and has 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 provided.

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:

First Class Functions and Higher-order Functions

Brewin# now supports first-class functions:

- You can pass functions/closures to other functions
- You can return functions/closures from other functions
- You can define variables of the "func" type which can hold functions or closures.
- You can store functions or closures in variables

Here's an example program:

```
func inc x:int int
  return + x 1
endfunc

func main void
 var func f             # f has a type of func and is a func variable
 assign f inc           # f holds the inc function!
 funccall f 10          # call inc thru f
 funccall print resulti # prints 11
endfunc
```

Lambdas and Closures

Brewin# now supports the creation of anonymous/lambda functions, including nested lambda functions and support for the capture of free variables (i.e., creating closures):

```
func create_lambda x:int func
  lambda y:int int     # defines a lambda/closure and stores in resultf
    var int z
    assign z + x y
    return z
  endlambda

  return resultf       # return our lambda/closure
endfunc

func main void
  var func f g
  funccall create_lambda 10   # create a lambda that captures x=10
  assign f resultf            # f holds our lambda's closure

  funccall create_lambda 100  # create a lambda that captures x=100
  assign g resultf            # f holds our lambda's closure

  funccall f 42
  funccall print resulti      # prints 52

  funccall g 42
  funccall print resulti      # prints 142
endfunc
```

Objects

Brewin# now supports the creation of objects which can hold an arbitrary set of member variables of all Brewin types (including function member variables and other object member variables). Brewin# doesn't support classes, just objects, like JavaScript. To add a member to an object, you just assign it (e.g., x.member = 10, like in Python). Brewin# doesn't require you to declare the member variables explicitly or specify their types.

```
func main void
 var object foo             # foo is an object!
 assign foo.x 5             # creates a member x in foo, e.g., foo.x,
                            # and sets its value to 5
 assign foo.x + foo.x 1     # increments foo.x
 funccall print foo.x
 assign foo.x "bar"         # redefines foo.x to be a string
 funccall print foo.x
endfunc
```

And you can even add member functions to your objects, as shown here:

```
# defines a function which will be used as a member function below
func foo i:int void
  assign this.val i    # sets the val member of the passed-in object
endfunc

func main void
  var object x
  assign x.our_method foo      # sets x.our_method to foo()

  funccall x.our_method 42     # calls foo(42)
  funccall print x.val         # prints 42
endfunc
```

Notice that our foo() function uses the "*this*" variable. Like C++'s *this* and Python's *self*, the *this* keyword is used to access the object being operated on. In the example above, *this* refers to object x.

Brewin# Language Detailed Spec

The following sections provide detailed requirements for the Brewin# language so you can implement your interpreter correctly. Other than those items that are explicitly listed below, all other language features must work the same as in the original Brewin++ language. As with Project #2, you may assume that all programs you will be asked to run will be syntactically correct (though perhaps not semantically correct). You must only check for the classes of errors specifically enumerated in this document, although you may opt to check for other (e.g., syntax) errors if you like to help you debug.

Type/Variable Changes

Function Variables

- You must now support the *func* type:

```
var func a b c
```

- The default value of function type is a dummy function that takes no parameter and immediately returns without doing anything. It is functionally equivalent to the following function:

```
func dummy void
endfunc
```

- Variables of the function type may be assigned to either regular functions or to lambdas/closures:

```
func foo void
  ...
endfunc

func main void
  var func a
  assign a foo
endfunc
```

Or:

```
func main void
  var int capture_me
  lambda a:int int
    return + a capture_me
  endlambda

  var func f
  assign f resultf      # see below for what resultf is!
endfunc
```

- Notice that you don't specify a detailed type signature for a function variable (e.g., one that specifies return types/parameters). A function variable may point at any valid function or lambda/closure. So the variable a in the first example above could be assigned to a function with two int arguments that returns a string, or a function with no arguments that returns an object, etc.

Object Variables

- You must now support the *object* type:

```
        var object x y z
```

- An object may be essentially represented as a dictionary that maps member variables to their values/types.
- The default value of object type is an empty object with no field populated.
- As we'll see, each object may have a unique set of members/methods, and members/methods are specified via assignment.

Parameter-passing/Return Changes

- You may now pass functions/lambas/closures as parameters to other functions; the parameter will have a type of *func*:

  ```
  func takes_a_function f:func void
    funccall f 10
  endfunc

  func foo x:int void
    funccall print x
  endfunc

  func main void
    funccall takes_a_function foo
  endfunc
  ```

  Note: Function parameters are passed by value, as with primitive types. But there is no `reffunc` type, which means you cannot pass functions by reference.

- You may now pass objects by object reference as parameters to other functions; the parameter will have a type of *object (which is really an object reference)*:

  ```
  func main void
    var object x
    assign x.member1 42
    assign x.member2 "blah"
    funccall foo x
  endfunc

  func foo q:object void
  ```

```
  funccall print q.member1
  assign q.member2 “bletch”  # mutates original x.member2 variable
endfunc
```

- You may now return functions/lambas/closures from functions - the returned function will be placed in resultf variable; the return type will be of type *func*:

```
func foo x:int int
  return + x 1
endfunc

func bar func
  return foo
endfunc

func main void
  var func f

  funccall bar
  assign f resultf        # resultf contains foo
  funccall f 10
  funccall print resulti  # prints 11
endfunc
```

- You may now return objects from functions - the returned object will be placed in resulto variable and is *returned by object reference* (i.e., a deep copy is NOT made of the returned object); the return type will be of type *object*. An object defined within a function and then returned from that function will have its lifetime extended until it is no longer referred to (just like Python objects).

```
func bar object   # the bar function returns an object
  var object x
  assign x.a 10
  assign x.b True
  return x
endfunc

func main void
  var object q

  funccall bar
  assign q resulto        # resulto contains x
  funccall print q.b      # prints True
endfunc
```

- Both resultf and resulto follow the same rules (e.g. scoping, default value, etc.) as with other three result variables defined in Brewin++.

Assignment and Expression Evaluation

- You may now use dot-notation to set/access members within objects:

  ```
  func main void
    var object x
    assign x.a 10
    assign x.b “salamander”
    funccall print x.a x.b
  endfunc
  ```

  Assignment of a member variable inside an object automatically creates a new member variable in the object. You do NOT declare variables explicitly within objects (to do so is a syntax error). You can assume that member variables follow the same naming conventions as typical variables.

- Assignment can be used to change the type of a member variable:

  ```
  func main void
    var object x
    assign x.a 10                     # a’s type is int
    assign x.a “salamander”           # a’s type changed to string
    funccall print x.a                # prints salamander
  endfunc
  ```

- Object members can be used in expressions, so long as the type of the member is compatible with the other operators/operands of the expression:

  ```
  func main void
    var object x
    assign x.a 10                     # x.a’s type is int
    var int y
    assign y 20

    assign y * x.a y
    funccall print y                  # prints 200
  endfunc
  ```

- Objects can contain other objects:

```
func main void
  var object x y z
  assign x.a 10

  assign y.my_member x
  assign z y.my_member

  funccall print z.a   # prints 10
endfunc
```

Think about this - it's super cool! Now you can create linked lists or trees in Brewin#!!

- You may create a method within an object by assigning a member variable to a function's name or a function variable:

```
func f i:int void
  ... # hidden for now
endfunc

func main void
  var object x

  assign x.some_method f          # x.some_method points at f()
  funccall x.some_method 52       # calls f(52)

  var func my_func
  assign my_func f                # my_func holds f

  assign x.second_method my_func   # x.second_method points at f()
endfunc
```

- You need to support only one level of dot-nesting, e.g.:

```
      funccall print z.a           # This MUST be supported
      funccall print y.my_member.a  # This is not supported and will
                                    # not be tested
```

- Accessing a member that doesn't exist must result in a name error at the time of execution:

```
func main void
  var object x
  assign x.a 10
  funccall print x.b   # results in a name error when this line runs
endfunc
```

- Trying to use the dot operator on a variable that is not of the object type results in a type error at the time of execution:

```
func main void
  var string xyz
  assign xyz.a 10       # results in a type error when this line runs
endfunc
```

- Assigning one object variable to another object variable copies the object reference, but does not make a deep copy of the object:

```
func main void
  var object o p

  assign o.x 10
  assign p o            # p now refers to the same object that o does.
  assign p.x 20
  funccall print o.x    # prints 20
endfunc
```

Function and Method Calls

- You may now pass a function variable to funccall:

```
func foo x:int void
  funccall print x
endfunc

func main void
  var func q
  assign q foo
  funccall q 5   # calls foo and prints 5
endfunc
```

- Attempting to call funccall on a variable that is not of the func type must result in a type error at the time of execution.

- You may now call a method using funccall and dot-notation (e.g., funccall circle5.get_area). Calling an object's method via dot notation will cause an implicit "this" *parameter* to be added to the top environment of the called method. The type of the *this* variable is *object* and it refers to the object on the left-hand-side of the method call:

  ```
  func f i:int void
    assign this.val i    # this refers to the x variable, below
  endfunc

  func main void
    var object xyz

    assign xyz.val 42
    assign xyz.method f

    funccall xyz.method 52
    funccall print xyz.val      # prints 52
  endfunc
  ```

- If you call a method without dot notation (see below), the language will NOT inject a hidden *this* parameter in the called function.

  ```
  func f i:int void
    assign this.val i    # Name error! "this" is undefined!
  endfunc

  func main void
    var object x

    assign x.val 42
    assign x.method f

    funccall f 52
    funccall print x.val        # never gets here
  endfunc
  ```

- "this" is considered a keyword in the language. See the "things we will not test you on" section for more on that.
- If you issue a funccall with a member variable that is not of the func type, this must result in a type error when the line is executed:

  ```
  func main void
    var object x
  ```

```
  assign x.val 42
  funccall x.val 52            # x.val is an integer, not a func
endfunc
```

- If you issue a funccall with an undefined member variable, this must result in a name error when the line is executed:

```
func main void
  var object x

  assign x.val 42
  funccall x.blah 52           # name error; x.blah doesn't exist
endfunc
```

Lambdas

- The Brewin# language now supports lambda functions with closures and variable capture. Lambda functions must be defined inside other functions (or nested inside of other lambdas). The syntax for defining a lambda is as follows:

```
      lambda param1:type1 param2:type2 … return_type
            # normal instructions here
            # this can do input, output, create new new variables, etc.
      endlambda
```

When a lambda line is executed, it must create a closure and then store the closure into the resultf variable, then continue executing on the line after the endlambda.

```
func main void
 var int capture_me
 assign capture_me 42

 lambda a:int int             # defines a lambda for int f(int)
  return + a capture_me       # captures the capture_me variable
 endlambda
 # resultf holds the closure created by the lambda

 var func f
 assign f resultf             # f now points to the closure
 funccall f 10                # calls our lambda function!
 funccall print resulti       # prints 52
endfunc
```

- A lambda captures all variables and parameters from all enclosing environments of the *current* function BY VALUE:

  ```
  func main void
   var int capture_me            # captured
   assign capture_me 42

   if > capture_me 10
     var int capture_me_too      # also captured
     assign capture_me_too 1000
     lambda a:int int
      return + + a capture_me capture_me_too
     endlambda
   endif

   funccall resultf 10           # resultf contains our closure
   funccall print resulti        # prints 1052
  endfunc
  ```

  When we say "capture by value," we mean that a deep copy of all captured variables is made, so any changes to the captured variables have no impact on the original variables. This is true for captured objects too - changes to a captured object within a lambda have no impact on the original variable (including objects referred to by those objects, etc.)! The following code illustrates what we mean when we say the lambda captures by value:

  ```
  func main void
    var int a
    var object o

    assign a 5
    assign o.x 10

    lambda void
      assign a + a 1
      funccall print a         # prints 6
      assign o.x 20
      funccall print o.x       # prints 20
    endlambda

    var func f
    assign f resultf
    funccall f
  ```

```
  funccall print a          # prints 5
  funccall print o.x        # prints 10
endfunc
```

- Nested lambdas must be supported - up to ten levels deep, e.g.:

```
func main void
  var int capture_me
  assign capture_me 1000
  lambda a:int func                   # outer lambda
   var int capture_me2
   assign capture_me2 10000
   lambda b:int int                   # nested lambda
    return + + + a capture_me b capture_me2
   endlambda
   return resultf
  endlambda

 var func f g
 assign f resultf       # f points at the outer lambda/closure
 funccall f 10
 assign g resultf       # g points at the inner lambda
 funccall g 42
 funccall print resulti       # prints 11052
endfunc
```

As you can see, a nested lambda captures all of the variables/parameters of its outer lambda(s).

- Lambda functions behave just like other functions, except they have a closure that captures free variables from the surrounding environment.

- The endlambda statement behaves just like the endfunc statement when running a lambda function.

- If there are more than one variable of the same name in the environment where the lambda is defined, the most-nested variable must shadow all variables of the same name with less-nested scope:

```
func main void
 var int capture_me
 assign capture_me 42
```

```
  if > capture_me 10
    var int capture_me
    assign capture_me 1000
    lambda a:int int
     return + a capture_me     # the captured capture_me’s value is 1000
    endlambda
  endif

  var func f
  assign f resultf
  funccall f 10
  funccall print resulti       # prints 1010
endfunc
```

Error Checking Requirements

Error checking stay the same as in Project #2 - you must only handle the errors detailed explicitly in this spec.

Things We Will Not Test You On

Things you will **not** have to worry about:
(Inherited from Project #2 and Project #1)

- Any syntax errors. You can assume that all programs will be syntactically correct.
- You will **never** have any nested double quotes inside strings in Brewin++. You **may** have single quotes inside double quotes.
- We will never call strtoint on a non-integral value.
- The program will always have valid indentation.
- You will never have variables with the same names as reserved keywords, e.g. `func`, `int`, etc.
    - note: `lambda`, `this`, and `object` are new reserved keywords in Brewin#
- You can assume that we will not manually define a result variable in any of our tests. For example, we will never have the statement `var int resulti` in any of our tests.
    - This includes `resultf` and `resulto`.

(Newly added in Project #3)

- We will never call print with arguments of object or function types (a variable or a literal).
- Different from Project #2, we will always call functions (including lambda functions) with the correct *number* of arguments that match the definition. Feel free to keep the checking behavior in Project #2.

- As a result, there is no need to support implicit partial application/currying. We will not test you on this.
- We will never create a circular object when testing (e.g. x.a = x).
- Lambda functions can appear in object methods. But you can assume that the lambda functions defined *within* object method functions do not use or capture the variable “this”.
- We will not create variables that have the same name of globally-defined functions, or shadow a globally-defined function name with a variable.
- We will never assign built-in functions to func variables; i.e. we will not test you with `assign f input, assign f strtoint, or assign f print`

Garbage Collection

You might not have thought about it, but all of those closures and objects you’re Brewin# programs are creating take up memory. However, you may simply rely upon Python’s garbage collection to free up discarded objects for you. There’s no need to build your own garbage collector.

Coding Requirements

Coding requirements are the same as Project #2.

Deliverables

For this project, you will turn in at least two files
- Your **interpreterv3.py** source file
- A readme.txt indicating any known issues/bugs in your program
- Other python source modules that you created to support your **interpreterv3.py** module (e.g., environment.py, type_module.py)

You **should not turn in intbase.py. Do not submit grader.zip. That is for instructors!**

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 updated the template GitHub repository: it now contains intbase.py, a copy of Carey’s **fully working solution for Project 2,** 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.

Similar to Project 2, we will give you your entire mark on submission; you can resubmit as many times as you'd like.

- 20% of the test cases are provided in the autograder repo.
- 80% of the test cases are hidden from you until grades are published. However, you'll see your score on these test cases on Gradescope.

New to Project 3: there are exactly 100 test cases total, and we've explicitly distributed the test cases across language features. In particular, missing any one feature (but correctly implementing the rest of the language) should result in you passing a good chunk of test cases.

Questions to Ponder

TBD. These might be helpful for exams! :)