Project #3: Brewin# Interpreter
CS131 Spring 2023
Due date: June 4th, 11:59pm

Introduction

The Brewin standards body (aka Carey) 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 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 have provided (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:

Default Field and Local Variable Values

Brewin# classes now support default field values and default local variable values, meaning that if a field or local variable definition omits an initializer value, the field's value will be initialized to the default value for the type (e.g., to 0 for ints, "" for strings, null for object references, etc.).

Here's an example program which shows both fields and local variables without initializers:

```
(class Dog
  ...
)

(class main
 (field Dog e)
 (method void main ()
  (let ((bool b) (string c) (int d))
    (print b)  # prints False
    (print c)  # prints empty string
    (print d)  # prints 0
    (if (!= e null) (print "value") (print "null")) # prints null
  )
 )
```

)

Of course, field and variable definitions can still have an initializer value if you like. It's just optional now.

Class Templates

Brewin# now supports class templates, so you can create generic classes that can be specialized to hold and process many different types of data. Here's a simple example that shows a templated class where the programmer has specified a single type to use for specialization; you can see how we instantiate objects with a concrete type below:

```
(tclass my_generic_class (field_type)
  (method void do_your_thing ((field_type x)) (call x talk))
)

(class duck
 (method void talk () (print "quack"))
)

(class person
 (method void talk () (print "hello"))
)

(class main
  (method void main ()
    (let ((my_generic_class@duck x null)
          (my_generic_class@person y null))
      # create a my_generic_class object that works with ducks
      (set x (new my_generic_class@duck))
      # create a my_generic_class object that works with persons
      (set y (new my_generic_class@person))
      (call x do_your_thing (new duck))
      (call y do_your_thing (new person))
    )
  )
)
```

Exception Handling

Brewin# now supports simple exception handling using try and catch blocks. A method may use the throw statement to throw an exception, and in Brewin# all exceptions are strings. A method may use a try statement to run a statement and catch any exceptions that occur by that statement or any sub-statements or calls made by that statement. Uncaught exceptions propagate up the call-chain until they are caught, or the program terminates - just like in other languages. Here's an example:

```
(class main
 (method void foo ()
   (begin
     (print "hello")
     (throw "I ran into a problem!")
     (print "goodbye")
   )
 )

 (method void bar ()
   (begin
     (print "hi")
     (call me foo)
     (print "bye")
   )
 )

 (method void main ()
  (begin
    (try
        # try running the a statement that may generate an exception
       (call me bar)
       # only run the following statement if an exception occurs
       (print "I got this exception: " exception)
    )
    (print "this runs whether or not an exception occurs")
  )
 )
)
```

The program above would print the following:

```
hi
hello
I got this exception: I ran into a problem!
this runs whether or not an exception occurs
```

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 Brewin++ language. As with Projects #1 and #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.

Default Field and Local Variable Values

Brewin# now supports default values for fields. Now, both of the following syntaxes are allowed:

```
(field type_name var_name initial_value)
```

and

```
(field type_name var_name)
```

e.g.

```
(field int x 10)  # with explicit initialization
(field bool y)    # with default initialization
```

Brewin# now supports default values for local variables defined in a let statement. Now, both of the following syntaxes are allowed:

```
(let ((type_name1 var_name1 initial_value1)
      (type_name2 var_name2 initial_value2) ...)
   (statement)
)
```

and

```
(let ((type_name1 var_name1)
      (type_name2 var_name2) ...)
   (statement)
)
```

e.g.

```
(let ((int a 5) (bool b true)) # with explicit initialization
 (statement)
)

(let ((int a) (bool b))      # with default initialization
 (statement)
)
```

Brewin# must now use the following default values for each type if an initialization value is omitted:

Type	Default Value
int	0
bool	False
string	""
object references	null

Class Templates

Brewin# now supports class templates, like those used in C++ but with much simpler syntax.

The syntax for defining a templated class as follows:

```
(tclass temp_class_name (parameterized_type1 parameterized_type2 ...)
  ...
)
```

The parameterized types can then be used anywhere a built-in type or user-defined type would be used, e.g.:

```
(tclass MyTemplatedClass (shape_type animal_type)
  (field shape_type some_shape)
  (field animal_type some_animal)
  (method void act ((shape_type s) (animal_type a))
    (begin
      (print "Shape's area: " (call s get_area))
      (print "Animal's name: " (call a get_name))
    )
  )
  ...
)
```

Once you have defined a templated class, you may use the class to define parameterized types anywhere normal types would be used in your program, e.g.: with return types, fields, parameters, local variables, and object instantiations.

The syntax for defining a fully-parameterized templated type as as follows:

```
temp_class_name@parameterized_type1@parameterized_type2@...
```

Here's an example

e.g.:

```
(class Square
  (field int side 10)
  (method int get_area () (return (* side side)))
)

(class Dog
  (field string name "koda")
  (method string get_name () (return name))
)

(class main
  (method void main ()
    (let ((Square s) (Dog d) (MyTemplatedClass@Square@Dog t))
      (set s (new Square))
      (set d (new Dog))
      (set t (new MyTemplatedClass@Square@Dog))
      (call t act s d)
    )
  )
```

)

Requirements:

- The type of an object reference must match exactly with the type of instances it refers to. For example, you can't point a MyTemplatedClass@int@string object reference at a MyTemplatedClass@string@bool object. Use of incompatible types must generate an error of type ErrorType.TYPE_ERROR.
- The main class may not be a templated class, and we will not test this case. Your code may have undefined behavior in this case.
- Templated classes may not be used in inheritance, and we will not test this case. Your code may have undefined behavior in this case.
- Variable names, method names, and class names may NOT contain @. We will not test your code against this case.
- Given two object references of a templated type, you may only compare those object references if they have the same parameterized types. For example, you can't compare a MyTemplatedClass@int@string object reference to a MyTemplatedClass@string@bool object reference using any of the comparison operators. Use of incompatible types must generate an error of type ErrorType.TYPE_ERROR.
- Specifying an invalid parameterized type name (e.g., one that is not a valid built-in type or a previously-defined class type) must generate an error of type ErrorType.TYPE_ERROR. For example, defining a variable with a type of MyTemplatedClass@tsring@bool should generate a type error.
- You may compare a templated object reference of any/all types to the null constant, e.g., (if (== s null) …)
- A templated class is guaranteed to have at least one parameterized type. We will not test you on cases where a templated class has no type parameters. Your code may have undefined behavior in this case.
- An attempt to use a templated class to define a type without fully specifying *all* type parameters must result in an error of type ErrorType.TYPE_ERROR. For example, using the MyTemplatedClass above, which takes two type parameters, in the following manner would generate type errors:

  ```
  (field MyTemplatedClass a)          # missing both parameterized types
  (field MyTemplatedClass@int a)      # missing the second parameterized type
  ```
- Brewin# does not allow nesting of templated types, and we will not test your interpreter against these cases, e.g.:

  ```
  (tclass class1 (type1) ...)
  (tclass class2 (type1 type2) ...)

  (class main
    (field class1@class2@bool@string)
    ...
  ```

)

- A templated class may call all appropriate methods and use all appropriate operators on its parameterized type(s), just like a template in C++. In other words, Brewin# templates are not like "generics" in Java but "templates" like in C++.
- If a templated object contains method calls or operations which are invalid for a specific parameterized type, then you must generate an error of type ErrorType.TYPE_ERROR **if and only if** such an invalid operation is actually performed at runtime. If an invalid operation is never performed on a value of such a type (even if the class contains methods that *could* perform an invalid operation on that value), then you do not need to generate an error - we will not test such a case.
- If there is a call to a parameterized class with a different type than the templated type (creating a method signature mismatch), then you must generate an error of type ErrorType.NAME_ERROR, similar to P2. Examples are below.

- For example, consider this totally legal Foo class which contains two methods which are incompatible for a single type parameterization (one method can operate on a type with a quack method, and the other clearly was meant to process integers):

```
(tclass Foo (field_type)
  (method void chatter ((field_type x))
    (call x quack)         # line A
  )
  (method bool compare_to_5 ((field_type x))
    (return (== x 5))
  )
)
```

If used with the following Duck class, and main method below:

```
(class Duck
 (method void quack () (print "quack"))
)

(class main
  (field Foo@Duck t1)
  (field Foo@int t2)
  (method void main ()
    (begin
       (set t1 (new Foo@Duck))      # works fine
       (set t2 (new Foo@int))       # works fine
```

```
            (call t1 chatter (new Duck)) # works fine - ducks can talk
            (call t2 compare_to_5 5)     # works fine - ints can be compared
            (call t1 chatter 10)  # generates a NAME ERROR on line A
          )
        )
      )
```

The first two calls to chatter and compare_to_5 will work just fine, because those methods use appropriate operations on each parameterized type (a Duck has a quack method, and an int can be compared against another int). However, the last method call to chatter must result in a NAME ERROR at the time the method call to quack is attempted, since t1 doesn't contain a method called "chatter" with an int parameter (since t1 is type Foo@Duck, it contains a version of "chatter" with a Duck parameter, not an int parameter).

Further Clarification on Name Error vs Type Error

Here are 3 examples of errors involving templates — the first one generates a name error, and the second + third ones generate type errors.

```
(tclass Foo (field_type)
  (method void chatter ((field_type x))
    (call x quack)))

(class Duck
  (method void quack ()
    (print "quack")))
(class main
  (field Foo@Duck t1)
    (method void main ()
      (begin
        (set t1 (new Foo@Duck))
        (call t1 chatter 5) #generates a name error
)))
```

Now, this generates a name error on the last line, since the parameterized class Foo@Duck doesn't contain a method with name chatter and formal parameter of type int.

```
(tclass Foo (field_type)
  (method void chatter ((field_type x))
    (call x quack))) #error generated here

(class Duck
  (method void quack ()
    (print "quack")))
(class main
  (field Foo@Duck t1)
    (method void main ()
      (begin
        (set t1 (new Foo@int)) #changed type of t1
        (call t1 chatter 5)
#generates a type error on the line above, mismatch between Foo@Duck
and Foo@int
)))
```

Now if we change the type of t1 to Foo@int, we get a type error, since t1 is defined as a field with type Foo@Duck.

Similarly, the following also generates a type error:

```
(tclass Foo (field_type)
  (method void compare_to_5 ((field_type x))
    (return (== x 5)) #== operator applied to two incompatible types
  )
)

(class Duck
  (method void quack ()
    (print "quack")))
(class main
  (field Foo@Duck t1)
    (method void main ()
      (begin
        (set t1 (new Foo@Duck))
        (call t1 compare_to_5 (new Duck)) #type error generated
)))
```

Since the == operator is applied to two incompatible types (Duck and int).

Exception Handling

Brewin# now supports limited exception handling for user-generated exceptions which are explicitly thrown by code (i.e., by a "throw" statement).

Like other languages, a thrown exception will immediately transfer control to the nearest/most nested exception handler in the call stack, e.g.:

- The exception will be caught first by an exception handler in the enclosing block,
- Then caught by an exception handler in its enclosing block if none was present in the innermost enclosing block,
- and so on, within the same method

If an exception handler that covers the exception isn't present in the same method, then the exception will immediately terminate the current method and be propagated to the caller of the method, and the process described above will be repeated recursively until either the exception is caught by an exception handler or the program terminates.

Similarly, an exception that happens during evaluation of an *expression* (e.g., *(+ 5 (call me some_method_that_throws_an_exception)*) must result in termination of the evaluation of that expression, and then the exception handling flow described above.

In Brewin#, when you throw an exception you must specify a string argument which communicates what went wrong. You would use the following syntax to throw an exception:

```
(throw <string expression>)
```

Where <string expression> can be a string constant, field, parameter, local variable or expression. So for example, the following would be valid throws:

```
(throw "foo")
(throw i_am_a_string_variable)
(throw (+ "foo" i_am_a_string_variable))
```

In Brewin# an exception handler takes the form of a *try* statement, using the following syntax:

```
(try
     (statement-to-try)   # run this statement always
     (catch-statement)    # only run this one if an exception occurs
)
... later code
```

The statement-to-try is always executed first. If this statement executes without throwing an exception then execution continues with the statement immediately following the end of the try statement (where it says "... later code"). On the other hand, if the statement-to-try or any of the statements it executes *does* throw an exception, then the catch-statement runs, followed by execution of the later code (if the catch-statement doesn't return from the method or throw a new exception).

The statement-to-try can be a single statement, a while statement, a begin statement, or a let statement, with arbitrarily-deep levels of nesting. The catch-statement can also be a single statement, a while statement, begin or let statement, with arbitrarily-deep levels of nesting.

If and when the catch statement is executed, a new variable called *exception* must be added into the catch statement's scope, and its value will be set to the exception string that was thrown by the throw statement. This allows the catch statement or any of its sub-statements to examine the thrown string to determine the cause of the exception. So, for example, the following would be valid a try statement:

```
(class main
  (method void bar ()
     (begin
        (print "hi")
        (throw "foo")
        (print "bye")
     )
  )
  (method void main ()
    (begin
      (try
       (call me bar)
       (print "The thrown exception was: " exception)
      )
      (print "done!")
    )
  )
)
```

This would print:

```
hi
The thrown exception was: foo
done1
```

The *exception* variable is only visible within the scope of the catch-statement, and is not accessible outside of this scope, e.g.:

```
(class main
  (method void main ()
    (begin
      (try
       (call me bar)
       (print "The thrown exception was: " exception)
      )
      (print "This should fail: " exception)  # fails with NAME_ERROR
    )
  )
)
```

The above code should generate an error of ErrorType.NAME_ERROR, since the *exception* variable is not visible outside of the catch block.

If an exception is thrown and there is no try statement to handle the exception in the current method, then the method will instantly terminate (all of its locals/parameters will go out of scope, and it will exit without a return value). This means that if the exception is thrown in a loop, or within a nested statement like a let or begin, all remaining statements in the loop, let or begin will be skipped and the function will immediately exit. Similarly, if the expression is thrown during a method call in an expression evaluation, further evaluation of the expression must be terminated, followed by the processing described in the sentence above. This holds recursively, so for example, if a statement within a let nested within a begin nested within a while loop throws an exception, and there is no try block covering this exception in the current method, execution will immediately terminate all statements in each of the nested blocks/loops before returning from the current method. For example:

```
(class main
 (method void foo ()
   (while true
     (begin
       (print "argh")
       (throw "blah")
       (print "yay!")
     )
   )
 )

 (method void bar ()
  (begin
     (print "hello")
```

```
      (call me foo)
      (print "bye")
    )
  )

  (method void main ()
    (begin
      (try
        (call me bar)
        (print exception)
      )
      (print "woot!")
    )
  )
)
```

This program prints:

```
hello
argh
blah
woot!
```

Other details:

- Brewin# programs are not allowed to define any methods, fields, local variables or parameters named "exception" and you will not be tested for these cases. Your program may have any behavior it likes for Brewin# code that does so.
- Your code must check to make sure that the parameter to the throw statement is of the string type. Attempting to throw other types of values (e.g., ints, bools, or objects) must result in an error of ErrorType.TypeError.
- Omitting the parameter to the throw statement should be considered a syntax error, and thus all test cases we provide you with will have a valid parameter (but it may not be a string!). You will therefore never have to deal with something like this: (throw)
- It is legal for a catch-statement or one of its sub-statements to throw an exception itself

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

- WE WILL TEST YOUR INTERPRETER ON ONLY THOSE SEMANTIC AND RUN-TIME ERRORS EXPLICITLY SPECIFIED IN THIS SPEC AND THE PROJECT #1 AND PROJECT #2 SPECS

Deliverables

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

- Your interpreterv3.py source file - the file MUST be named interpreterv3.py
- You may submit as many other supporting Python modules as you like (e.g., *class.py*, *method.py*, …) which are used by your *interpreterv3.py* 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., environment.py, type_module.py)
- You MUST NOT modify our intbase.py file since you will NOT be turning this file in. If your code depends upon a modified intbase.py file, this will result in a grade of zero on this project.

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

We will be grading your solution on **Python 3.11**. **Do not use any external libraries that are not in the Python standard library. We strongly recommend that you do not use or import the sys module in your submitted code as it may cause issues with our autograder.**

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 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.