Project #2: Brewin++ Interpreter

CS131 Spring 2023

Due date: May 21st, 11:59pm

A quick note on undefined behaviour:

Undefined behavior refers to a situation where the execution of some code is unpredictable.

This **does not** mean that the code in question will always fail or perform some unsafe operations; it also **does not** mean that expected behavior is impossible.

If we state that your code may have undefined behavior in a particular situation, don't worry about your code not matching the exact behavior of barista. Having different behavior during different runs on the same inputs is even okay if the spec defines the operation as causing undefined behavior.

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 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 #1 solution, or by modifying the Project #1 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

Brewin++ now implements static typing[1], meaning all fields and parameters have types, and methods 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, an object can't be passed to a Boolean parameter, etc).

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

```
(class main
 (method int add ((int a) (int b))
    (return (+ a b))
 )
 (field int q 5)
 (method void main ()
  (print (call me add 1000 q))
 )
)
```

[1] Even though Brewin++ is an interpreted language, by adding variable definitions 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.

Default Return Values from Functions

In Brewin++, if a method has a non-void return type then it must return a value. If the statements within the method'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 method's return type upon the method's completion (e.g., 0 for ints, false for Booleans). This way, all methods return some value even if they don't explicitly have a return statement to do so. So, for example:

```
(class main
  (method int value_or_zero ((int q))
     (begin
       (if (< q 0)
         (print "q is less than zero")
         (return q) # else case
       )
     )
   )
  (method void main ()
    (begin
      (print (call me value_or_zero 10))  # prints 10
      (print (call me value_or_zero -10)) # prints 0
    )
  )
)
```

In the above value_or_zero method, if the number passed in is less than zero then the print statement will execute and the function will terminate without explicitly returning a value. Since the method didn't explicitly return a value in this case, the interpreter will automatically return the default value for the type for the method, which for integers is zero. On the other hand, if the user were to pass in a positive number, then the else clause of the if statement will run and the method will return the passed in value.

Local Variables

Brewin++ now supports local variables, which must be defined as part of a "let" statement. A let statement is like a begin statement, except the first item in the let statement is a block of one or more variable definitions with types and initial values specified:

```
(class main
 (method void foo ((int x))
     (let ((int y 5) (string z "bar"))
        (print x)
```

```
           (print y)
           (print z)
        )
    )
    (method void main ()
      (call me foo 10)
    )
   )
```

The local variables defined within a let statement are only visible to the statements nested within the let. Notice that the let statement can have many sub-statements.

Inheritance

Brewin++ now supports simple inheritance. A derived class may have its own methods/fields, and may override the methods of the base class just as with other languages. Here's an example:

```
(class person
  (field string name "jane")
  (method void set_name ((string n)) (set name n))
  (method string get_name () (return name))
)

(class student inherits person
  (field int beers 3)
  (method void set_beers ((int g)) (set beers g))
  (method int get_beers () (return beers))
)

(class main
  (field student s null)
  (method void main ()
    (begin
      (set s (new student))
      (print (call s get_name) " has " (call s get_beers) " beers")
    )
  )
)
```

Polymorphism

Brewin++ now supports polymorphism. As with languages like C++, you can substitute an object of the derived class anywhere your code expects an object of the base class. Here's an example:

```
(class person
  (field string name "jane")
  (method void say_something () (print name " says hi"))
)

(class student inherits person
  (method void say_something ()
    (print "Can I have a project extension?")
  )
)

(class main
  (field person p null)
  (method void foo ((person p)) # foo accepts a "person" as an argument
    (call p say_something)
  )
  (method void main ()
    (begin
      (set p (new student))  # assigns p, which is a person object ref
                             # to a student object. This is allowed!
      (call me foo p)        # passes a "student" as an argument to foo
    )
  )
)
```

Each method call must be directed to the most overridden method associated with the object, just as with other OOP languages. For example, in the program above, the call to say_something in the foo method should be directed to student's say_something and not person's say_something method. We'll see later that Brewin++ also introduces a "super" object reference, which allows an object to call a method defined in one of its superclasses (if that method has been overridden in the derived class).

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 v1 language. As with Project #1, 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.

Static Typing

Brewin++ now implements static typing with NO implicit or explicit conversions between primitive types. This requires you to implement the following:

All fields must have a type specified

Syntax:

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

e.g.

```
(field int x 10)
(field person p null)
```

Requirements:

- A type must either be a primitive type (e.g., int, bool, string) or a class type (which may only be a class defined *above* the current field definition - there's no need to handle the case for classes defined below)
- You must support the ability to have a field with a type that's the same as the class the field is defined in, enabling use cases like linked lists (e.g. a Node class, with a field of type Node called next)

  ```
  (class Node
  ```

```
    (field Node next null)
    ...
  )
```

Methods must now have a return type and types for each parameter

Syntax:

```
(method return_type func_name ((type1 param1) (type2 param2) ...)
   (method statement)
)
```

Notice that each parameter and its type is now enclosed in parentheses.

e.g.,

```
(method string foo ((string a) (string b)) (return (+ a b)))
```

Type Checking

You must perform the following type checks in your interpreter.

Field Initialization Type Checks

You must check that the initializer value in a field definition is compatible with a variable's type, and generate an error of type ErrorType.TYPE_ERROR if this is violated:

```
(field int x 52)       # OK!
(field person p null)  # OK assuming a person class is defined
(field int x "foo")    # ERROR: Must generate an ErrorType.TYPE_ERROR
```

Assignment and Comparison Type Checks

You must check that during an assignment of a variable to a value, the variable and value have compatible types. Similarly, you must check that during comparison of two variables/values, the variables/values must have compatible types. The following rules may be used to determine type compatibility:

- In an assignment or comparison of primitive variables/values, both must have exactly the same type (e.g., no comparison of bool and ints are allowed)

- In an assignment or comparison of object references:
    - You may assign an object reference of type X to an object of type X
    - You may assign an object reference of type B to an object of type D, where the D class is derived from the B class
    - You may compare an object reference of type X to another of type X
    - You may compare an object reference of type B to an object reference of type D, where the D class is derived from the B class
    - You may assign/compare object references of any type to null

If any of the above are violated, then your interpreter must generate an error of type ErrorType.TYPE_ERROR.

The following code snippets show valid assignments/comparisons (note the full class definitions are not shown for brevity):

```
# the assignments are valid since both sides are ints!
(field int x 0)
(method void foo ((int param1) (int param2))
  (begin
    (set param1 param2)
    (set x param2)
  )
)

# the assignments are valid since both sides are of person type!
(field person pf null)
(method void foo ((person p1) (person p2))
  (begin
    (set p1 p2)
    (set pf p2)
    (set p1 (new person))
  )
)

# these assignments are valid if the student class inherits from the
# person class
(field person pf null)
(method void foo ((person p) (student s))
  (begin
    (set p s)
    (set pf p)
    (set pf s)
    (set p (new student))
  )
```

```
)

# the comparison is valid since both object references refer to
# person objects
(method void foo ((person p1) (person p2))
  (if ((== p1 p2)
    (print "same object")
  )
)

# the comparison is valid since student is derived from person
(method void foo ((person ref1) (student ref2))
  (if (== ref1 ref2)   # valid if student inherits from person
    (print "same object")
  )
)

# null can be compared to an object reference of any type
(method void foo ((dog r))
  (if (== r null)
    (print "invalid object")
  )
)

# all obj references can be set to null
(method void foo ((dog r))
  (set r null)
)

# the assignment is valid because the returns_int method returns an
# int value, and i is an int variable
(method int returns_int () (return 5))
  …
(method void foo ((int i))
  (set i (call me returns_int))
)
```

These are invalid and must generate an error of type ErrorType.TYPE_ERROR:

```
# invalid since the parameters are of different types
(method void foo ((int param1) (string param2) …)
  (set param1 param2)
)
```

```
# invalid since we can't set a subtype variable to refer to a
# supertype object
(method void foo ((student param1) (person param2) …)
  (set param1 param2)
)

# the comparison is invalid since person and dog unrelated classes
(method void foo ((person ref1) (dog ref2) …)
  (if (== ref1 ref2)
    (print "same object")
  )
)

# even though student/prof might both be be derived from person,
# neither is a superclass of the other
(method void foo ((student ref1) (professor ref2) …)
  (if (== param1 param2)
    (print "same object"))
)
```

Parameter Passing Type Checks

The interpreter must check that all arguments passed to a method have compatible types with the types of the formal parameters:

- All primitive types passed to a method must match the types of the formal parameters exactly
- You may pass an object of type X to a method that has a parameter of type X
- A derived object D may be passed to a method that accepts a base object of type B (e.g., you can pass a student object to a method that has a parameter of type person)

If a parameter of the wrong type is passed, then your interpreter must generate an error of type ErrorType.TYPE_ERROR. Here are some examples:

The following code snippets show valid passing of values (note the full class definitions are not shown for brevity):

```
# valid since type of variable q is int and type of parameter x is int
(class main
  (field int q 30)
  (method void foo ((int x)) (print x))
  (method void main ()
    (call me foo q)
  )
)
```

```
# valid since type of variable pers is person and type of parameter p is person
(class main
  (field person pers null)
  (method void ask_person_to_talk ((person p)) (call p talk))
  (method void main ()
    (begin
      (set pers (new person))
      (call me ask_person_to_talk pers)
    )
  )
)
```

```
# valid since type of variable pers is a subclass of person and type of parameter p
# is person (assumes student derived from person)
(class main
  (field student s null)
  (method void ask_person_to_talk ((person p)) (call p talk)))
  (method void main ()
    (begin
      (set s (new student))
      (call me ask_person_to_talk s)
    )
  )
)
```

These are examples of invalid parameter passing and must generate an error of type ErrorType.TYPE_ERROR:

```
# invalid since type of variable q is bool and type of parameter x is int
(class main
  (field bool q true)
  (method void foo ((int x)) (print x))
  (method void main ()
    (call me foo q)
  )
)
```

```
# invalid since type of variable pers is person and type of parameter p is dog
(class main
  (field person pers null)
  (method void ask_dog_to_bark ((dog d)) (call d bark))
  (method void main ()
    (begin
      (set pers (new person))
      (call me ask_dog_to_bark pers)
    )
  )
)
```

# invalid since while both student and professor are subtypes of person, student is not
# a subtype of professor

```
(class main
  (field student stud null)
  (method void ask_prof_to_talk ((professor p)) (call p talk)))
  (method void main ()
    (begin
      (set stud (new student))
      (call me ask_prof_to_talk stud)
    )
  )
)
```

Returned Value Type Checks

The interpreter must check that all values returned from a method must have a compatible type with the method's return type, and that no values are returned from a method with a void return type:

- A method with a primitive return type of P may return a value of type P, but not a value of any other type
- A method with a return type of class X may return an object of class X
- A method with a return type of some base class B may return an object of type D, if D is derived from B (e.g., you can return a student object from a method that has a return type of person)
- A method that has a return type of any class type X may return null
- A method with a return type of void must not return any value (it may use the return statement, but it may not specify a return value)

The following code snippets show valid methods using return (note the full class definitions are not shown for brevity):

# valid because the foo method has a return type of int, and returns an int value

```
(class main
  (method int foo () (return 5))
  (method void main () (print (call me foo)))
)
```

# valid because the foo method has a return type of person, and returns an person
# object

```
(class main
  (method person foo () (return (new person)))
  (method void main () (call me foo))
)
```

# valid because the foo method has a return type of person, and returns a student
# object (where we assume student is derived from person)

```
(class main
  (method person foo () (return (new student)))
  (method void main () (call me foo))
)
```

# valid because null may always be returned from a function that has a class return type

```
(class main
  (method person foo () (return null))
  (method void main () (call me foo))
)
```

```
# valid because the foo method has a void return type and uses the
# return statement without specifying a value
(class main
  (method void foo ((int q))
    (if (== q 0)
      (return)
      (print "q is non-zero")
    )
  (method void main () (call me foo 5))
)
```

The following code snippets show invalid methods using return, and must generate an error of ErrorType.TYPE_ERROR:

# invalid because the foo method has a return type of int, and returns an bool value

```
(class main
  (method int foo () (return false))
  (method void main () (print (call me foo)))
)
```

```
# invalid because the foo method has a return type of person, and returns a dog
# object (where dog is not derived from person)
(class main
  (method person foo () (return (new dog)))
  (method void main () (call me foo))
)

# invalid because the foo method has a return type of student, and returns a professor
# object (where we assume student is not derived from professor)
(class main
  (method student foo () (return (new professor)))
  (method void main () (call me foo))
)

# invalid because the foo method has a return type of student, and returns a person
# object (where student is derived from person)
(class main
  (method student foo () (return (new person)))
  (method void main () (call me foo))
)

# invalid because the foo method has a void return type but tries to return a value
(class main
  (method void foo () (return 5))
  (method void main () (call me foo))
)
```

Default Return Values for Methods

If a method doesn't explicitly return a value using a return statement, then when the method finishes running, the interpreter must (implicitly) return the default value for the method's return type. This includes cases where every statement in a method runs without executing a return statement, or when a return statement runs, but it does not have an argument that specifies the value to return, e.g. (return).

- Methods with an int return type must return 0 by default
- Methods with an bool return type must return false by default
- Methods with a string return type must return the empty string ("") by default
- Methods with a class return type must return null by default

For example:

# since foo doesn't explicitly return a value, the interpreter returns a value of zero
# for the function once it finishes. Thus the print statement in main prints 0.
```
(class main
  (method int foo () (print "hi"))
  (method void main () (print (call me foo)))
)
```

# prints out empty string
```
(class main
  (method string foo () (return))
  (method void main () (print (call me foo)))
)
```

# prints out true, then prints false
```
(class main
  (method bool foo ((bool q))
    (if q
      (return)  # returns default value for bool which is false
    )
    (return true)
  )
  (method void main ()
    (begin
      (print (call me foo false))  # prints true
      (print (call me foo true))   # prints false
    )
)
```

Local Variables

Brewin++ now supports local variables, which must be defined as part of a "let" statement. A let statement is like a begin statement, except the first item in the let statement is a block of zero or more variable definitions with initial values specified. Its syntax is as follows:

```
(let ((type1 var_name1 init_value1) (type2 var_name2 init_value2) …)
   (statement1)
   (statement2)
   …
   (statement2)
)
```

As you can see, each variable definition specifies a type, the variable's name, and its initial value. The local variables are only visible to the statements within the let block, and they go out of scope once the let block completes. If a variable defined in a let block has the same name as a field, a parameter to the method, or a variable defined in an outer let block then the variable defined in the innermost let block hides or "shadows" those other variables. Here is an example:

```
(class main
 (method void foo ((int x))
   (begin
     (print x)                              # Line #1: prints 10
     (let ((bool x true) (int y 5))
       (print x)                            # Line #2: prints true
       (print y)                            # Line #3: prints 5
     )
     (print x)                              # Line #4: prints 10
   )
 )
 (method void main ()
   (call me foo 10)
 )
)
```

Notice how in the above example, the let block defines a boolean variable named x. Within the let block's statements, references to the x variable shadowed the outer x parameter variable, so the code prints out **True** when asked to print out x's value on line #2. Once the let block completes, printing out x's value on line #4 results in a value of 10 being printed, since the reference to x now again references the parameter.

Any attempts to access a variable defined in a let outside of the let block must result in an error of type ErrorType.NAME_ERROR:

```
(class main
 (method void foo ()
   (begin
     (let ((int y 5))
       (print y)        # this prints out 5
     )
     (print y)  # this must result in a name error - y is out of scope!
   )
 )
 (method void main ()
   (call me foo)
 )
)
```

Additional Requirements:

- If a let statement defines two variables with the same name, then your interpreter must generate an error of type ErrorType.NAME_ERROR.
- A let statement must always have at least one statement to execute, but may have more than one statement just like a begin statement

Inheritance

Brewin++ now supports inheritance. The general syntax is as follows:

```
(class derived_class_name inherits base_class_name
  # class fields and methods defined as usual
)
```

Example:

```
(class person
  (field string name "anonymous")
  (method void set_name ((string n)) (set name n))
  (method void say_something () (print name " says hi"))
)

(class student inherits person
  (field int student_id 0)
  (method void set_id ((int id)) (set student_id id))
  (method void say_something ()
    (begin
     (print "first")
     (call super say_something)  # calls person's say_something method
     (print "second")
    )
  )
)

(class main
  (field student s null)
  (method void main ()
    (begin
      (set s (new student))
      (call s set_name "julin")   # calls person's set_name method
      (call s set_id 010123456) # calls student's set_id method
```

```
        (call s say_something)    # calls student's say_something method
      )
    )
  )
```

Details:

- A derived class inherits all of the methods and fields of all of its base class(es)
  - Instantiation of a derived object will automatically initialize all of the fields from every class in the hierarchy of the derived object
- You may have an unlimited number of levels of inheritance, e.g., organism → animal → mammal → human → cyborg
- As with Brewin v1, fields in each class are private, meaning that a subclass has NO access to the fields of its superclass(es); a derived class must call the methods of the base class in order to access/modify fields in the base class
- Since all methods are public, a derived class contains and publicly exposes all of the methods of its superclasses.
- The derived class may add new methods, redefine existing methods (override the implementation of the method from a superclass), or add methods of the same name with a different number of parameters (overloading)
- Calling a method of an object will always run the most-derived version of that method, just like in C++ or Python
- If a derived method M wishes to call the superclass version of method M which has the same return type/parameters, it must use the following syntax:

  (call super method_name arg1 arg2 …)

- You may assume that we will never test your code against a case where a method defined in the base and re-defined in the derived classes have the same name and parameters but a different return type.
- *Overloading* of methods defined in superclasses is allowed in subclasses, so for example, this is legal:

```
(class foo
 (method void f ((int x)) (print x))
)
(class bar inherits foo
 (method void f ((int x) (int y)) (print x " " y))
)

(class main
 (field bar b null)
 (method void main ()
   (begin
     (set b (new bar))
     (call b f 10)      # calls version of f defined in foo
     (call b f 10 20)   # calls version of f defined in bar
   )
 )
)
```

Polymorphism

Brewin++ now supports polymorphism. As with languages like C++, you can substitute an object of the derived class anywhere code expects an object of the base class. Here's an example:

```
(class person
  (field string name "jane")
  (method void say_something () (print name " says hi")
  )
)

(class student inherits person
  (method void say_something ()
    (print "Can I have an extension?")
  )
)

(class main
  (field person p null)
  (method void foo ((person p)) # foo accepts a "person" as an argument
    (call p say_something)
  )
  (method void main ()
    (begin
      (set p (new student))  # Assigns p, which is a person object ref
                             # to a student obj. This is polymorphism!
      (call me foo p)        # Passes a student object as an argument
                             # to foo. This is also polymorphism!
    )
  )
)
```

Details:

Assuming we have three classes B, D, and DD, where B is the base class, D is derived from B, and DD is derived from D (we could also have had a DDD derived from DD, etc).

- You may assign a field, local variable or parameter of type B to refer to an object of type D, DD, DDD, etc. Similarly you may assign a variable of type D to refer to an object of type DD, DDD, etc. And so on.
- You may pass an object of type D, DD, DDD, etc. to a method that expects a parameter of type B. You may pass an object of type DD, DDD, etc. to a method that expects a parameter of type D, etc.
- Assume that a method M is defined in B and then overridden in D (or DD, or DDD, etc). Calling M through an object reference of type B will call the most-derived version of that method associated with the actual object being referred to (i.e., D's version of the method)
- You may NOT pass an object of type B to a method that accepts a parameter of type D, DD, etc. Doing so must generate an error of type ErrorType.TYPE_ERROR.

Things We Will and Won't Test You On

Your may assume the following when building your interpreter:

- WE WILL NOT TEST YOUR INTERPRETER ON SYNTAX ERRORS OF ANY TYPE
- WE WILL TEST YOUR INTERPRETER ON ONLY THOSE SEMANTIC AND RUN-TIME ERRORS EXPLICITLY SPECIFIED IN THIS SPEC.
  (Any errors not mentionned in this spec should retain their behaviour from part 1.)

Deliverables

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

- Your interpreterv2.py source file - the file MUST be named interpreterv2.py
- You may submit as many other supporting Python modules as you like (e.g., *class.py*, *method.py*, …) which are used by your *interpreterv2.py* file.
- A readme.txt indicating any known issues/bugs in your program (or, “all good!”)
- 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.**

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. **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.**

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. The TAs have developed a tool called barista (barista.fly.dev) that lets you test any Brewin code and provide the canonical response. Remember to choose the right version of brewin to use as the barista setting.