Project #2: Brewin++
CS131 Fall 2022
Due date: Nov 8th, 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 #1 solution, or by modifying the Project #1 solution that we 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:

Variable and Typing Changes

Brewin++ now implements static typing[1] for all variables. All variables now must be defined before they are used (like in C++), and a type must be specified in each definition. In addition, the language must ensure that all variable assignments and expressions have compatible types; no implicit conversions are allowed.

Here's an example program:

```
func main void
 var int b c d       # b,c, and d are integer variables
 assign b 5

 var bool e          # e is a boolean variable
 assign e "foobar"   # this would be a type error!
```

[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.

```
endfunc
```

Scoping Changes

Brewin++ now implements lexical scoping for all variables, just like you'd see in C++. Here's an example program:

```
func main void
  var int a
  assign a 5
  if > a 0
    funccall print a    # prints 5
    var string a        # legal: shadows our original a variable
    assign a "foobar"
    funccall print a    # prints foobar
    var bool b
  endif
  funccall print a      # prints 5
  funccall print b      # Name error: b is out of scope
endfunc
```

Function Changes

Brewin++ now supports parameters for user-defined functions, and allows arguments to be passed to these functions by value or by reference. Functions also now must have an explicit return type. To access the return value from a function, you no longer refer to the *result* variable, but refer to *resulti* for integers, *resultb* for booleans, and *results* for strings.

Here's an example program:

```
# Equivalent to: bool absval(int val, int& change_me)
func absval val:int change_me:refint bool
  if < val 0
    assign change_me * -1 val
    return True
  else
    assign change_me val
    return False
  endif
endfunc
```

```
func main void
  var int val output
  assign val -5
  funccall absval val output
  funccall print "The absolute value is: " output
  funccall print "Did I negate the input value? " resultb
endfunc
```

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.

Variable Changes: Definition and Typing

- Variables now must be defined before they are used (like C++), and a type must be specified in the definition
    - Valid variable types are: int, bool, string
    - More than one variable may be defined at a time on a line
    - The syntax is:
      var type $variable_1$ $variable_2$ … $variable_n$

  Examples:

  ```
  var int alpha beta gamma  # defines alpha, beta and gamma as ints
  var string delta # defines delta as a string
  ```

  If an undefined variable is referred to in any part of your program (e.g., you try to print its value out, use it in an expression, or assign it a value and it was not explicitly defined as a formal parameter or a variable), you must report an error when interpreting the line by calling the InterpreterBase.error() method with an error type of ErrorType.NAME_ERROR.

  Note: If you create a variable of unknown type, then you must report an error when interpreting the line by calling the InterpreterBase.error() method with an error type of ErrorType.TYPE_ERROR.

- The value given to a newly-defined variable, whose value has not yet been explicitly assigned, is the default value for the type, e.g.:
    - int: 0
    - bool: False
    - string: “”
- The language must ensure that all variable assignments and expressions have compatible types; no implicit conversions are allowed. If the types in an expression or assignment are not compatible, you must generate an error when interpreting the line by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.

  Examples:

```
var int a          # defaults to zero
var bool b         # defaults to false
assign a b         # type error; no conversion allowed between bool to int
assign b 1         # type error, no conversion allowed between int to bool
assign a * 3 b  # type error, booleans can't be multiplied with integers
```

Variable Changes: Scoping

- Variables are scoped to the function in which they are defined, and are not global (and therefore not visible to other functions). When a function ends, all variables defined in a function go out of scope and their lifetime ends, e.g.:

```
func foo void
      # error - blah is NOT known to foo(), only to main()
      funccall print blah
endfunc

func main void
      var int blah
      funccall foo
endfunc
```

- Variables are scoped to the block in which they are defined (and are visible in nested blocks within the block where they are defined); when a block ends, all variables defined in that block go out of scope and their lifetime ends.

```
func main void
  var int a
  assign a 5
  if > a 0
    funccall print a    # this is valid; a is visible to inner block
```

```
    var string b        # b is scoped to the if-statement block
  endif
  funccall print b      # Name error: b is out of scope
endfunc
```

- Duplicate variable definitions within the same block are not allowed.

  ```
  func main void
    var int a
    assign a 42
    var int a # this is illegal!
  endfunc
  ```

  If a variable is redefined in the same block, you must report an error when interpreting the line by calling the InterpreterBase.error() method with an error type of ErrorType.NAME_ERROR.

  Note: Defining a variable with the same name as an argument to a function counts as a redefinition, and should be handled in the same way.

- An inner/nested block may define a variable of the same name as a variable in an enclosing block, and that definition will shadow (hide) the outer definition until the end of the current block, when the inner variable disappears, e.g.:

  ```
  func main void
    var int a
    assign a 5
    if > a 0
      funccall print a    # prints 5, since a is visible from outer scope
      var string a        # new a variable shadows our original a
      assign a "foobar"
      funccall print a    # prints foobar
    endif
    funccall print a      # prints 5, since inner-foobar is out of scope
  endfunc
  ```

- The scope of all results returned by a function must be scoped to the top level of the calling function, e.g.:

  ```
  func bar int
        # code to compute some value
        return some_val
  endfunc
  ```

```
func foo void
        if True
                funccall bar                    # resulti is set upon return from bar
                funccall print resulti          # valid since resulti is in scope
        endif
        funccall print resulti                  # also valid since resulti has top-level scope
endfunc
```

- resulti, resultb, or results must only be defined if they are explicitly set by a called function that returns a value. So this would be invalid:

  ```
  func foo void
          funccall print resulti  # we never called a function, so we can't refer to resulti
  endfunc
  ```

  You must report an error when interpreting the line by calling the InterpreterBase.error() method with an error type of ErrorType.NAME_ERROR.

  Note: Calling a function which returns one type should not affect the result variables of the other types. For example, calling a function which returns an integer, should not affect resultb or results.

Function Changes: Parameter Passing

- Functions now may be defined with zero or more typed parameters which may be passed by value or by reference. Functions must also have an explicit return type. The syntax for a function header is:

  func funcname var_1:$type_1$ var_2:$type_2$ … var_n:$type_n$ return_type

  Where parameter types may be: int, bool, string, refint, refbool, refstring
  Where the return types may be: int, bool, string, void

  We guarantee that there will be no spaces in between the parameter name, the colon, and the type name.

  Example function headers:

  ```
  func foo xyz:int pdq:refbool void
  func bar int
  func bletch pqr:refstring void
  ```

- Type compatibility:

    - You may pass an int/bool/string variable to a function that accepts a refint/refbool/refstring parameter, respectively.
    - You may pass an refint/refbool/refstring variable to a function that accepts a refint/refbool/refstring parameter, respectively.
    - You may pass an refint/refbool/refstring variable to a function that accepts an int/bool/string parameter, respectively.
    - If an actual parameter's type and a formal parameter's type are not compatible, you must generate an error when interpreting the function call line by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.
- Changes to parameters:
    - Parameters may have their value changed within a function. If the parameter is of a reference type, then the assignment will modify the original referred-to variable's value. If the parameter is passed by value, then the assignment will change the local variable associated with the parameter and must not modify the caller's variable that was passed in.
        - Note: You must consider the case where we pass a variable x to a function that accepts a reference y, then pass y to another function that accepts a reference z, etc. Modifying parameter z must result in the original variable, x, being modified.
    - It is **legal** to pass a constant value to a function that accepts a reference. In such a case, if the reference parameter is reassigned, this change will only be reflected within the current function, since the original constant value can't be changed.
    - If a function is called with more/less parameters than its definition, then you must generate an error when interpreting the function call line by calling InterpreterBase.error() with a type error of ErrorType.NAME_ERROR.

Function Changes: Returning Values

- All functions must have an explicit return type of int, bool, string or void, e.g.:

```
func foo arg:int void
endfunc

func bar a:string b:bool int
endfunc
```

- When returning an expression/value from a function, the type of that value being returned must be consistent with the return type of the function.
    - If the type being returned is incompatible with the return type of the function, then you must generate an error when interpreting the return line by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.

- If the return command is used in a function that returns an int, bool or string (not void), and a return value/expression is not specified, then the default value for the type (0 for int, False for bool, “” for string) must be returned to the caller.

  For example, the following are invalid:

```
func foo void
      return 5          # error
endfunc

func bar int
      return “blah”     # error
endfunc
```

  For example, the following are valid:

```
func foo string
      return            # ok; returns “” default value
endfunc

func foo int
      return 5          # ok
endfunc

func foo arg:refint int
      return arg  # refint and int are compatible
endfunc
```

  Note: Specifying no return statement is equivalent to a default return

- While in Brewin v1, results were always returned in a variable called “result”, in Brewin++ results are delivered via three separate result variables:
  - Upon returning from a function f() that returns an integer result, the return value of f() must be placed in a variable named resulti
  - Upon returning from a function f() that returns a boolean result, the return value of f() must be placed in a variable named resultb
  - Upon returning from a function f() that returns a string result, the return value of f() must be placed in a variable named results

- All results (resulti, resultb, results) must scoped to the top-level scope of the calling function, to enable use cases like this:

```
func foo arg:int void
```

```
 if > arg 5
  funccall input "Enter your name: "
 else
  funccall input "Enter your dog's name: "
 endif
 # notice how results is referred to outside the block where
 # we called input to get the name.
 funccall print "You entered: " results
endfunc

func main void
 funccall foo 5
endfunc
```

Expressions and Assignment Changes

- During expression evaluation, the interpreter must validate that the variables' types (as specified in their definition) are compatible with their operators, and with the types of other variables/constants within the expression.

  Note: It is acceptable to use a refint/refbool/refstring variable in an expression that otherwise is operating on int/bool/string items, e.g.:

```
func foo arg:refint void
        var int b
        assign b * arg 5          # arg is a refint, 5 is an int, this is OK
endfunc
```

- During assignment, the interpreter must validate that the variable having its value changed has a compatible type with the expression/value that it's being set to, e.g.

```
var string a
var  int b
assign b a   # this is an error, since b is not a string
```

  Note: it is acceptable to assign a refint variable to an int, and visa-versa, e.g.:

```
func foo arg:refint void
        assign arg 5     # arg is a refint, 5 is an int, and arg can be assigned to int
        var int b
        assign b arg     # b is an int, arg is a refint, and be can be assigned to arg
endfunc
```

- If any types within an expression or an assignment are invalid, then you must generate an error when interpreting the line by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.

Error Checking Requirements

Error checking stay the same as in Project #1 - you must only handle the errors detailed explicitly in this spec.

Things you will **not** have to worry about:

- 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. You may have variables named as `resulti/resultb/results` however.
- You can assume that we will not redefine a result variable in any of our tests. For example, we will never have the statement `var int resulti` in any of our tests.

Coding Requirements

Coding requirements are the same as Project #1, with the exception of:

- You must name your interpreter source file interpreterv2.py
- **We will not increase the default recursion limit in Python. Your code must not hit the recursion limit.**
    - To compensate, we will provide you the “most recursive” test case; if it passes this test case, it should work properly on the hidden cases.

While this is not required, we have a handful of suggestions to make your code play nicely with the autograder:

- Don’t include code that runs globally; wrap your main routine in an if-guard
- Don’t use exit() or any other code that could abruptly terminate your program; instead, use our provided error tools in intbase
- Don’t modify signal handlers (if you don’t know what this means, you’re probably not doing it)

Deliverables

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

- Your interpreterv2.py source file
- A readme.txt indicating any known issues/bugs in your program
- Other python source modules that you created to support your interpreterv2.py module (e.g., environment.py, type_module.py)

You should not submit intbase.py; we will provide our own. **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 1,** 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.

New from Project 1: we now give you your entire mark on every submission.

- 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.

Questions to Ponder

TBD. These might be helpful for exams! :)