Project #1: Brewin Interpreter
CS131 Fall 2022
Due date: Oct 23rd, 11:59pm

Introduction

In this project, you will be implementing a simple interpreter for a new programming language, called Brewin. You'll be implementing your interpreter in Python. This project is the first of three - in the later CS131 projects, you'll be adding new features to your language based on the concepts we learn in class.

Once you successfully complete this project, your interpreter should be able to run simple Brewin programs and produce a result, for instance it should be able to run a program that computes the factorial of a number the user enters (see below).

Brewin v1 Language Introduction

Here is a simple program in the Brewin language, with line numbers added for clarity (the Brewin language does NOT use line numbers in its source code):

```
000 # Our first Brewin program!
001 func main        # our main function
002    funccall input "Enter a number: "
003    assign str_val result
004    funccall strtoint str_val
005    assign n result
006    funccall factorial
007    funccall print result
008 endfunc
009
010 func factorial   # compute a factorial
011  assign f 1
012  while > n 1
013    assign f * f n
014    assign n - n 1
```

```
015  endwhile
016  return f
017 endfunc
```

Here's a description of the above program:

- Line #0: This line has a comment on it, indicated by the #
- Line #1: Defines a function called main; all Brewin programs must have a main function. This is where the interpreter starts our program.
- Line #2: This makes a function call to the input function with the prompt "Enter a number: ". The input function is a built-in language function that inputs a *string* from the keyboard. The input function, like all functions in Brewin, returns a result by setting a global variable named "result" with the inputted value.
- Line #3: This is an assignment statement. It assigns a variable called str_val to result (which is a built-in variable that contains the user-inputted string from line #2). Notice that we didn't have to define the variable "str_val" to use it. In Brewin v1, just assigning a variable creates a new global variable for us.
    - The str_val variable is created on this line, and is a global variable. It's now visible across all functions in our program.
- Line #4: This makes a function call to the strtoint function, which converts its argument from a string to an integer, and places the result in the "result" global variable, overwriting its previous value. The strtoint function is a built-in language function.
- Line #5: Creates a new global variable n, and assigns it to the integer result in the "result" variable.
    - The n variable is created on this line, and is a global variable. It's now visible across all functions in our program.
- Line #6: This calls the factorial function. Notice that we don't pass any arguments. The Brewin v1 language does not need function arguments, since all variables are global.
- Line #7: This makes a function call to the print function, which prints its argument's value to the screen. In this case, we're printing the result of the factorial function. The print function is a built-in language function.
- Line #8: This denotes the end of the main function. Notice that it is tabbed out the same number of spaces as the "func main" definition.
- Line #10: This defines a factorial function.
- Line #11: This creates a new global variable called f and sets its value to 1.
- Line #12: This is a while loop. The expression following "while" must be a boolean expression. All expressions in Brewin are in prefix notation, e.g. a + b → + a b
    - Notice that the loop refers to the global n variable, which was defined in our main function. This is legal in Brewin v1, since all variables are global.
- Line #13: This line assigns f to f*n.
- Line #14: This assigns n to n-1.
- Line #15: This denotes the end of the while loop. Notice that it is tabbed out the same number of spaces as the "while" part of the loop on line #12.

- Line #16: This returns the value of the variable f as the result from the function. Return causes the value, f, to be copied into the global “result” variable, so it can be used on line #7.
- Line #17: This denotes the end of the factorial function.

Let’s distill some interesting facts about the Brewin v1 language from our program:

- Formatting:
  - Brewin requires that statements be indented like in Python. You must have at least one space of indentation for each new block.
  - Comments in Brewin begin with a # sign.
  - All statements use spaces as separators - no parentheses, commas, semicolons, etc.
- Variables:
  - All variables are global, and once defined, are visible to all functions and have a lifetime that extends through the end of the program. Variables do NOT disappear when a function ends, and are visible to their callers.
  - Variables are assigned via the “assign” statement.
  - Variables are defined when they are first assigned to a value.
  - You don’t specify a type for a variable. You just assign it to a value. So the variable doesn’t have a type, but the value it refers to does have a type. So Brewinv1 is a “dynamically typed” language.
- Functions:
  - All user-defined functions are defined via the “func” keyword.
  - All functions are called with the funccall keyword.
  - User-defined functions have no arguments.
  - Built-in functions like input, strtoint, and print do take arguments, and those must either be variables or constants (expressions are NOT allowed).
  - A function may return a value. If it does so, a new global variable called “result” is created (if it doesn’t already exist) and it is set to the returned value.
- Expressions:
  - Expressions are written in prefix notation, e.g. + 5 * 6 3, or > n 1
  - An expression may refer to constants, variables, or both.
  - Expressions may evaluate to strings, booleans, or integers.
- Loops:
  - The while loop accepts a single boolean expression, which it uses to decide whether to run its body.
  - The loop may run one or more statements inside its body.

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

How Does an Interpreter Work?

Think back to our first day of class - I showed an interpreter on a slide! Here's the basic structure of an interpreter class:

```
class Interpreter {
public:
  void run(const vector<string>& program) {
     program_statements_ = program;
     reset_all_variables();
     ip_ = locate_first_line_of_main_function();  // figure out where to start interpreting
     terminated_ = false;

     while (!terminated_) {
        interpret();   // runs the statement on line ip_ and advances to the next statement
     }

private:
  int interpret() {
    // parse the current statement into tokens (e.g., "funccall print 5" → "funccall", "print", "5")
    vector<string> tokens = split_into_tokens(program_statements_[ip_]);

    // Handle each possible statement type
    if (tokens[0] == "funccall")  process_funccall(...);
    else if (tokens[0] == "while") process_while(...);
    else if (tokens[0] == "endwhile") find_corresponding_while_statement_and_jump_back(...);
    …

    ip_ = figure_out_next_ip_to_run_after_this_statement_();
  }

  vector<string> program_statements_;  // The program is stored here
  int ip_; // Specifies the line number (instruction pointer) of the next statement to run
  map<string,Value> variable_name_to_value;  // stores global variables & their values
  bool terminated_;  // Have we reached the end of the program?
}
```

More complicated languages require extensive grammars, parsing frameworks, creation of abstract syntax trees and/or bytecode, etc. However, given the simplicity of the Brewin language, you can do very simple parsing (with a couple of special cases to handle strings) making your interpreter far simpler.

Brewin v1 Language Spec

The following sections provide detailed requirements for the Brewin v1 language so you can implement your interpreter correctly.

Formatting

- Indentation is similar to Python indentation.
    - An if statement and its corresponding endif and/or else must be preceded by the same number of spaces.
    - A while statement and its corresponding endwhile must be preceded by the same number of spaces.
    - All statements inside of an if-block, an else-block, or a while-block must be spaced out at least one additional space.
    - You must not use tabs, only spaces, for indentation.
- Comments begin with a # and extend to the end of the current line. A # character inside of a string must be ignored, e.g., the # in “foo # bar” does not indicate not a comment
- All tokens (e.g., language keywords, operators, constants, variable names) are case sensitive.
- All tokens are separated by one or more spaces. There are no commas, parentheses, colons, semicolons, etc. in the language. Be careful, there can be spaces inside of strings, e.g. “this is a string”! So:
    - funccall print “this is a string”

  should be tokenized into:

  funccall
  print
  ”this is a string”

  and not:

  funccall
  print
  “this
  is
  a
  string”
- Each statement is confined to a single line; a given statement must not span multiple lines.
- You may ignore all trailing spaces at the end of lines, or lines that just have a comment.

Variables

- Variable names are case sensitive.
- Variables must begin with a letter and are composed of letters, numbers and underscores.

- All variables are global; once defined, they exist for the rest of the program and are visible from all functions in the program, including those called the function where they are first defined.
- Variables are not defined explicitly; the first assignment defines a new variable and sets its value.
- Variables do not have a particular type - they can be assigned to a value of any type, and may be reassigned to a value of another type over time.
- If an undefined variable is referred to in any part of your program other than assignment (e.g., you try to print its value out), you must report an error by calling the InterpreterBase.error() method with an error type of ErrorType.NAME_ERROR.
- Brewin has a special variable named “result” which is set by a return statement in order to communicate a result to a calling function. This variable does not exist until the first return statement returns an expression, so any attempt to access the result variable prior to execution of a return statement should also fail with an error type of ErrorType.NAME_ERROR.

Constants

- You can have integer constants, string constants enclosed in “double quotes”, and boolean constants (True/False); there are no floating-point numbers.
- Integer constants may be positive or negative (e.g., -5 with no spacing between the minus sign and the first digit), and have any valid integer value representable by a python integer.
- Strings may contain spaces and # characters inside them, so be careful! :)

Expressions

- An expression may be any of the following:
    - A constant (int, string, boolean)
    - A variable
    - An arithmetic expression, with any of the following operators: +, -, *, /, %, <, >, <=, >=, !=, ==
        - The arithmetic operators must yield an integer result
        - The comparison operators must yield a boolean result
    - A string expression, with any of the following operators: + (for concatenation), ==, !=, <, >, <=, >=
        - The + operator must yield a string result
        - All comparison operators compare strings lexicographically
        - The comparison operators must yield a boolean result
    - A boolean expression, with any of the following operators: !=, ==, & (logical AND), | (logical OR)
        - All operators yield a boolean result
- All expressions are represented in prefix notation with spaces separating each token in the expression.

- Notice that expressions cannot contain function calls. Function call statements must be on a line of their own.
- The types of all values used in an expression must be compatible, e.g., you can't add a string and an integer, or compare a boolean and an integer. If the types do not match, you must generate an error when interpreting the line by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.
- If an operator is not compatible with an operand, then you must generate an error when interpreting the line by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.
- You are not responsible for handling things like divide by zero, integer overflow, integer underflow, etc. Your program may behave in an undefined way if these conditions occur.

Here are some valid example expressions, assuming *var1*, *var2, and str1* are valid variables:

- var1
- 32
- "this is # a string"
- -5
- > var1 32
- == var1 - var2 5
- & > var1 5 <= var2 3
- == str1 "foobar"

Functions

User-defined Functions

- Function names are case sensitive.
- Functions must begin with a letter and are composed of letters, numbers and underscores.
- A function is globally visible to all other functions, above and below it in the file.
- Functions may optionally return a value with the return statement.
- If an undefined function is referred to in any part of your program, you must report an error when the function is called during interpretation by calling the InterpreterBase.error() method with an error type of ErrorType.NAME_ERROR.
- A function must be defined with the following case-sensitive syntax:

```
func funcname
        # one or more statements
endfunc
```

- Every Brewin program must have a function called main
    - Interpretation begins on the first line of the main function and ends once the last line of main has executed, or a return statement in main is executed
- To call a user-defined function, use the following syntax:

```
funccall function_name
```

where function_name must be the name of a valid defined function, e.g., foo:

```
func foo
        …
endfunc

func main
        funccall foo
endfunc
```

Brewin v1.0 does not support first-class functions, so you must only use the name of an explicitly-defined function (like foo) or a built-in function (like print, input) after the funccall keyword.

- You may assume that it is illegal to define two or more functions with the same function name; you do not need to check for violation of this rule in your interpreter.
- You may assume that it is illegal to redefine built-in functions (e.g., print, input); you do not need to check for violation of this rule in your interpreter.

Built-in Functions

You must support the following three built-in functions in your implementation:

input

- Syntax: funccall input <constant or variable to use as a prompt> <constant or variable to use as a prompt> …
- Example usage:
    - This code:
        - assign name “Paul”
        - funccall input name “, please input a number:”
    - prints:
        - Paul, please input a number:
        - <cursor flashing here waiting for input>
- Requirements:
    - The input command concatenates all of its arguments together (without spaces in between the arguments) and then uses our provided InterpreterBase.output() to print the concatenated string. Then it inputs a string from the user by calling our provided InterpreterBase.get_input() method. Finally, the function sets the global “result” variable to the text the user typed (the text the user types in is returned as a string, even if it is just a number).

    - Your implementation MUST use our InterpreterBase.output() method to output the prompts to the user (so we can do automated testing). Our output() method will automatically append a newline after the prompt for you.
    - Your implementation MUST use our InterpreterBase.get_input() method to actually read input from the user (so we can do automated testing).
    - You must not add any additional spaces, newlines, etc. to your output, so:
        - funccall input 5 True “foobar”
        - should print *5Truefoobar* before prompting the user

print

- Syntax: funccall print <constant or variable> <constant or variable> …
- Example usage:
    - assign value 5
    - funccall print “here is your value: “ value
- Requirements:
    - The print command must construct a string with the specified constant(s)/variable(s), with no added spacing or newlines and print it using our InterpreterBase.output() method.
    - You may not pass a full expression (e.g., + 3 var) to print; you must only pass variables or constants as parameters
    - Your implementation MUST use our InterpreterBase.output() method to actually output data to the user (so we can do automated testing).

strtoint

- Syntax: funccall strtoint <constant or variable>
- Example usage:
    - assign s “1234”
    - funccall strtoint s
    - assign n result # n holds 1234 now
- Requirements:
    - The strtoint function must convert the specified string into an integer. It then sets the global “result” variable to the resulting integer.
    - You may not pass a full expression (e.g., + str1 str2) to strtoint; you must only pass variables or constants as parameters.
    - The function must check to make sure that the passed variable/constant is a string. If not, it must generate an error by calling InterpreterBase.error() with ErrorType.TYPE_ERROR.

Assignment

Assignment in Brewin v1 has two roles:

1. To create a new global variable and assign it an initial value.

2. To reassign a global variable to a new value.

The syntax of assignment is: assign <variable> <expression>

Example usage:

- assign foo 1
- assign bar “hello there!”
- assign foo + foo 3 # foo = foo + 3
- assign bar > foo 5 # bar = (foo > 5)
- assign foo result # take the last returned value from a function and set foo’s value to it

Returning Values

The return command immediately terminates a function, causing it to return to its caller (or terminate the program, in the case of the main function). A function may:

- Not have a return statement at all, in which case no value is returned from the function and no change is made to the global “result” variable (see below for more on the result variable). In this case, a function ends when it hits the endfunc statement.
- Have a return statement that has no expression, in which case the return statement simply terminates the function but no value is returned, and no change is made to the global “result” variable.
- Return the value of an expression, in which case the global “result” variable is updated with the value of the returned expression.

The syntax of returning is: return [optional expression]

The return statement assigns the value of the global variable named “result” to the value of the expression, if one is present. If the “result” variable does not yet exist, the return statement will create it first before assigning it. If no expression is specified, then the function simply returns with no change to the global “result” variable.

Example syntax:

- return 5
- return “foo”
- return + 5 bar
- return > bar 6
- return == bletch “foo”
- return # no value

Example program:

```
000 # A simple program showing how to use return statements
001 func main
002  funccall factorial
003  assign bar result  # bar gets a value of 7
004  funccall returns_nuthin
005  funccall print result  # result still has a value of 7
006 endfunc
007
008 func factorial
009  return 7
010 endfunc
011
012 func returns_nuthin
013 endfunc
```

Control Structures

If/Else Statements

If statements behave just like in other languages. Brewin supports and if/endif clause, as well as if/else/endif, but NOT if/elseif:

The syntax of an if statement is:

```
if <expression>
  # one or more statements run, iff expression is true
endif
# following statements, executed immediately if expression is false
```

Or:

```
if <expression>
  # one or more statements run, iff expression is true
else
  # one or more statements, iff expression is false
endif
# following statements
```

Requirements:

- The else and endif statements must be tabbed out the same number of spaces as the if statement they're associated with.

- If the expression evaluated by the if-statement is not a boolean, then you must generate an error when interpreting the line by calling InterpreterBase.error() with a type error of ErrorType.TYPE_ERROR.

While Statements

While statements behave just like those in other languages.

The syntax of a while statement is:

```
while <expression>
  # one or more statements run, iff expression is true
endwhile
# following statements, executed immediately if expression is false
```

Requirements:
- The endwhile statement must be tabbed out the same number of spaces as the while statement it's associated with.
- If the expression evaluated by the while-statement is not a boolean, 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

Your interpreter must meet the following requirements when it comes to error checking:

- You may assume that all programs that we present to your interpreter will be *syntactically* well-formed and not have any syntax errors. That said, if you do add checks for syntax errors, you're much more likely to save debugging time.
- You may NOT assume that all programs presented to your interpreter will be *semantically* correct, and must, at a minimum, address those errors that are explicitly called out in this specification, via a call to InterpreterBase.error() method. Make sure to report the offending instruction pointer for each error correctly, as our testing framework will use this to validate your interpreter's correctness.
    - Remember, the first line of the program is line number 0 (not 1).
- You must report an error the moment is detected during execution of an interpreted Brewin program. You must not detect errors prior to interpretation. That means that if a program has two errors, you must detect the first one that actually runs during interpretation.

Coding Requirements

You MUST adhere to the following coding requirements for your program to work with our testing framework, and thus to get a grade above a zero on this project:

- Your Interpreter class MUST be called Interpreter
- You must derive your interpreter class from our InterpreterBase class:

```
class Interpreter(InterpreterBase):
  …
```

- Your Interpreter class constructor (__init__()) must start with these two lines in order to properly initialize our base class:

```
def __init__(self, console_output=True, input=None, trace_output=False):
    super().__init__(console_output, input)   # call InterpreterBase's constructor
```

  The console_output parameter indicates where an interpreted program's output should be directed. The default is to the screen. But when we run our test scripts, we'll redirect the output for evaluation. You can just pass this field onto InterpreterBase's constructor and otherwise ignore it.

  The input parameter is used for our testing scripts. It's the way we pass input to your program in an automated manner. You can just pass this field onto InterpreterBase's constructor and otherwise ignore it.

  The trace_output parameter is used to help in your debugging. If the user passes True in for this, your program should output whatever debug information you think would be useful to the screen. For example:

```
class Interpreter(InterpreterBase):
  …
  def interpret_statement(self):
    print(f"{ip_}: {program_statements[ip_]}")
    … # your code to interpret the statement on the specified line number
```

- You must implement a run method in your Interpreter class. It must have the following signature:

```
def run(self, program):
  …
```

Where the second parameter, *program*, is an array of strings containing the Brewin program text that we want to interpret, e.g.:

```
program = [“func main”, “  funccall print “hello world!””, “endfunc”]

interpreter = Interpreter()
interpreter.run(program)
```

- When you print output (e.g., *funccall print “blah”*) you must use the InterpreterBase.output() method, to ensure that your program’s output can be evaluated by our test scripts. By default, our output method will display all output to the screen, but during automated grading we will redirect the output for testing.
- When you get input from the user (e.g., for a *funccall input* statement) you must use the InterpreterBase.get_input() method, to ensure that our test input can be passed in by our test scripts. By default, our get_input method will prompt the user via the keyboard.
- To report errors (e.g., typing errors, syntax errors, name errors) you must call the InterpreterBase.error() method with the specified error:

```
def interpret_statement(self, line_number, statement):
  …
  if cant_find_variable(v):
    super().error(ErrorType.NAME_ERROR,f”Unknown variable {v}”)
```

- We define all of the keywords for the Brewin language in our InterpreterBase class, e.g.:

```
# constants
FUNC_DEF = 'func'
ENDFUNC_DEF = 'endfunc'
WHILE_DEF = 'while'
ELSE_DEF = 'else'
ENDWHILE_DEF = 'endwhile'
ENDIF_DEF = 'endif'
ASSIGN_DEF = 'assign'
FUNCCALL_DEF = 'funccall'
```

  You should use these constants in your interpreter source file rather than hard-coding these strings in your code.
- You must name your interpreter source file interpreterv1.py
- Try to write self-documenting code with descriptive function and variable names and use idiomatic Python code.
- 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.

Deliverables

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

- Your interpreterv1.py source file
- A readme.txt indicating any known issues/bugs in your program
- Other python source modules that you created to support your interpreterv1.py module (e.g., environment.py, type_module.py)

You should not submit intbase.py; we will provide our own. **You should not submit a .zip file**.

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.

Questions to Ponder

TBD. These might be helpful for exams! :)