

# Project 5

COMP 443 – Programming Languages  
Spring 2016

**Due Tuesday, May 3 by 11:55 pm**

## Overview

For this project, you will be writing an interpreter for the Grove language.

You will be working in a group of 2 or 3 people for this project. All of the work on this project should be your team's own work. Do not show code to anyone outside your group, nor look at anyone else's code for this project. You can discuss general concepts with classmates (e.g., How do I create a object when I have a string for its type?), but not how they pertain to this project.

## Learning Objectives

Completing this project will give you experience with Python, including...

- Objects
- Duck typing
- Introspection

## Program Specifications

The ideas for this project build upon parts of the Calc interpreter that we built in class.

When errors occur with the parsing or evaluating of expressions, your program should throw a `GroveError`, defined as follows:

```
class GroveError(Exception):  
    def __init__(self, *args, **kwargs):  
        Exception.__init__(self, *args, **kwargs)
```

The command

```
raise GroveError("my message")
```

will create and raise the exception.

**Raising the `GroveError` exception is important**, because it is how I verify that you are catching invalid commands.

## Language

The Grove language is given on the last page of this document. This section explains the aspects of the language that differ from the Calc language we discussed in class.

- Name is more general than in Calc. A valid Name must start with an **alphabetic** character (including `_`), followed by zero or more **alphanumeric** characters (also including `_`). This is taken from Python 2's definition, which states the same idea in BNF:

[https://docs.python.org/2/reference/lexical\\_analysis.html#identifiers](https://docs.python.org/2/reference/lexical_analysis.html#identifiers)

- Expr no longer supports subtraction, but there are two new kinds of expressions:
  - **String literals.** The rule `<Expr> ::= "<Token>"` means that the following are valid expressions (**including the quotes**):

- `"hello"`
- `"a.b.c"`

whereas these are not:

- `"a b c"`
- `"a\"b"`

You should add a `StringLiteral` class to your available parse tree nodes. It should evaluate to the string itself. For example, `StringLiteral("hi").eval() == "hi"`

- **Method calls.** A method call expression starts with the keyword `call`, followed by open parenthesis, a Name, another Name, zero or more Arguments, and closing parenthesis.
  - The first Name is the name of an object in our global variables table. If you evaluate the call and that name does not exist in the variables table, raise a `GroveError`.
  - The second Name is the method name. You should check that the specified object has the method by using introspection. If the method is not defined, raise a `GroveError`.
  - The `"<Expr>*" syntax in the language specification indicates zero or more expressions come between the method name and the closing )".  
To parse the arguments, after handling the object Name and method Name, run a while loop:  
while the next token is not )" and there are still more tokens to parse:  
    parse the next token(s)  
    verify that the result is an Expr  
    append the Expr to a list of arguments (to be evaluated later)`
  - The arguments for these method calls are standard, positional arguments (i.e., not like `name=value` arguments or default arguments). If you have a list of values and you want to split them up so they are separate arguments to a Python function call, you can use the `*`operator to do so. See <https://docs.python.org/2/tutorial/controlflow.html#unpacking-argument-lists> for an example.

- The Addition operation should be modified from the Calc language, since Grove supports more than just integers. At runtime (i.e., when evaluating the expression), throw a `GroveError` if the two values being added do not have the same type.

- Stmt has changed substantially

- The `"set"` keyword denotes a variable assignment, as in the Calc language. However, there are now two kinds of right-hand-sides to accept. The first is an Expr, as in the Calc language.

The second is `"new" <Name> or "new" <Name>.<Name>`

It allows for statements like

```
new list
```

or  
`new calendar.Calendar`

A "new" statement should create a new Python object of the specified type, which is then assigned to the variable name on the left-hand-side of the set statement. For simplicity, our language does not support constructor arguments.

- The "quit" and "exit" keywords should each exit your interpreter when "evaluated." You can use Python's `sys.exit()` to do so.
- The "import" keyword allows access to existing Python modules. The evaluation of a Stmt with the "import" keyword should import the specified module so it is available for later commands. Use `importlib.import_module` to do this. Note that `importlib.import_module` returns the module, which you will need to store in the `globals()` dictionary. To figure out the key you should use, look at how `globals()` changes when you run a normal import statement in Python, like `import os`

## Interpreter

Once you have the classes defined for your parse tree nodes, you are ready to create them and evaluate them.

Write a **parse function** that **takes a string and returns the root of a parse tree**.

That **root should have an eval() method** that "runs" the command, producing the behavior described in the "Language" section above. If the command is an expression, the `eval()` method should return the value of the expression.

One of your Python files should be called **grove.py**. When run as a script (i.e., when the `__name__` is `"__main__"`), that file should enter a **read-evaluate-print-loop** until the user enters a quit or exit command. In your loop, you should do the following:

- Prompt the user for input with `"Grove>> "`
- Read the line that the user enters
- Parse the line
- Evaluate the command
- If the command is an expression, print the result to the screen. However, if the result is `None`, do not print anything.
- Repeat

If there is a `GroveError` with either the parsing or evaluation, you should **catch it in your loop** and print out an error message for the user before prompting for the next command.

## Written Question

In a comment at the top of `grove.py`, answer the following question [5 points]:

Is your Grove interpreter using a static or dynamic type system? Briefly explain what aspects of the interpreter make it so.

## Testing

There is a test suite on the course website to help you verify your interpreter.

- `no_parse.txt` is a list of commands, one on each line, that should raise `GroveErrors` when you try to parse them. [16 points]
- `no_eval.txt` is a list of commands, one on each line, that should parse correctly, but raise `GroveErrors` when you try to evaluate them. [16 points]
- `singles.txt` is a list of expressions, one on each line, that should parse correctly and evaluate to the respective values in `singles_answers.txt`. Each line can be evaluated independently of the others. [15 points]
- `single_stmts.txt` is a list of statements that should parse and evaluate correctly. Each statement can be evaluated independently of the others. [12 points]
- `test1.txt` (and similar) are lists of commands that should parse and evaluate correctly when run in order. The corresponding `answer1.txt` contains the output that your program should produce for `test1.txt`. [36 points]

The test suite also includes two drivers to run the tests:

- `check_errors.py` checks the `no_parse.txt` and `no_eval.txt` cases.
- `check_valid.ps1` is a Windows PowerShell script to run the other tests. Details are found in comments in the file. Run "powershell" from a Windows command prompt to open PowerShell (which will look very similar to the command prompt). Then you can run `check_valid.ps1`

Each driver will print error messages if there are tests that fail.

## Grading

Each test case should match exactly in order to receive credit for that test case. A "test case" is one line from a testing file, with the exception of the `test?.txt` files. Each of those `test?.txt` files is a single test case, for which you receive full credit or no credit.

## Handin

Zip your `.py` files and submit the zip file on myGCC.

## The Grove Language

Supports importing Python modules, creating objects, and calling methods.

```
<Command>      ::= <Expr>      | <Stmt>

// Expressions evaluate to a value
<Expr>          ::= <Num>        | <Name>
                  | "+" "(" <Expr> ")" "(" <Expr> ")"
                  | "<Str>"
                  | "call" "(" <Name> <Name> <Expr>* ")"

<Str>           ::= any string not containing whitespace or quotes

// Numbers are constant, non-negative integers
<Num>           ::= string of one or more digits

// Names are variable names, module names, method names, etc.
<Name>          ::= a string of one or more alphanumeric
                    characters, starting with an alphabetic character
                    For us, alphabetic includes _

// Statements do not evaluate to a value
// Instead, they have some side effect
<Stmt>          ::= "set" <Name> "=" <Expr>
                  | "set" <Name> "=" "new" <Name>
                  | "set" <Name> "=" "new" <Name> "." <Name>
                  | "quit"
                  | "exit"
                  | "import" <Name>
```