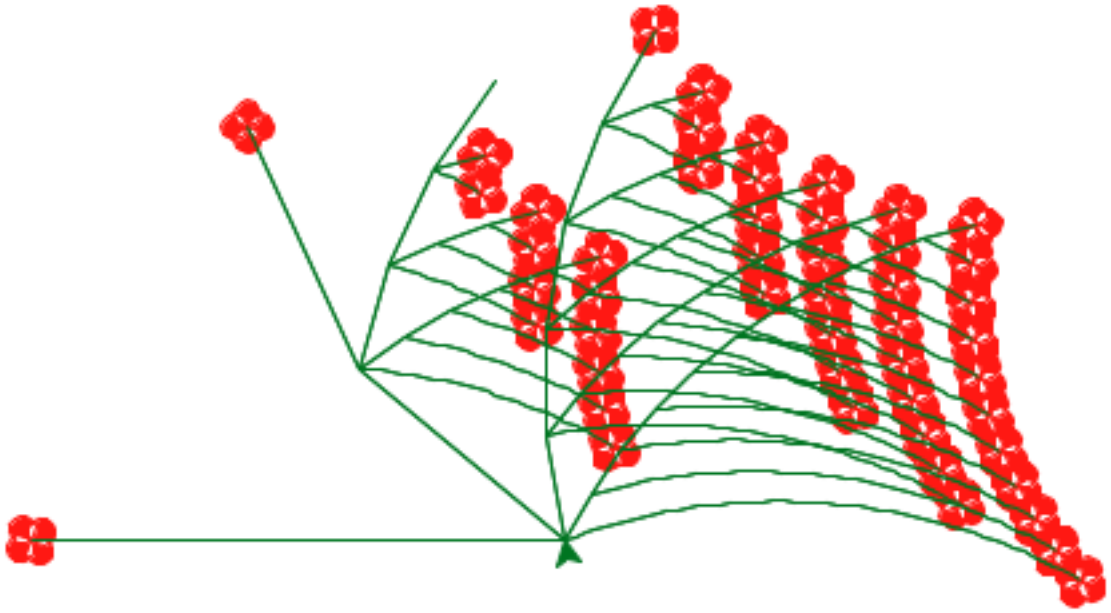
Project 4: A Logo Interpreter



Eval calls apply,
which just calls eval again!
When does it all end?

Introduction

In this project, you will develop an interpreter for the Logo language. As you proceed, think about the issues that arise in the design of a programming language; many quirks of languages are the byproduct of implementation decisions in interpreters and compilers.

You will also implement some small programs in Logo, including the `count_change` function that we studied in lecture. Logo is a simple but powerful functional language. You should find that much of what you have learned about Python transfers cleanly to other programming languages as well. To learn more about Logo, you can read Brian Harvey's textbooks online for free.

This project concludes with an open-ended graphics contest that challenges you to produce recursive images in only a few lines of Logo. As an example of what you might create, the picture above abstractly depicts all the ways of making change for $0.50 using U.S. currency. All flowers appear at the end of a branch with length 50. Small angles in a branch indicate an additional coin, while large angles indicate a new currency denomination. In the contest, you too will have the chance to unleash your inner recursive artist.

This project includes several files, but all of your changes will be made to the first three: `logo.py`, `tests.lg`, and `contest.lg`. You can download all of the project code as a zip archive.

`logo.py`	The Logo evaluator
`tests.lg`	Logo examples and expected output for your interpreter

`contest.lg`	A place to write your contest entry
`logo_parser.py`	The Logo parser
`logo_primitives.py`	Defines primitive Logo procedures via the Python Library
`logo_test.py`	A testing framework for Logo
`buffer.py`	A `Buffer` is a list that tracks an indexed position
`ucb.py`	Utility functions for 61A

Logistics

This is a three-part project. As in the previous projects, you'll work in a team of two people, person A and person B. In each part, you will do some of the work separately, but most questions can be completed as a pair. Both partners should understand the solutions to all questions.

After completing the first part, you will be able to evaluate primitive procedures with constant arguments. In the second part, you will add variables and user-defined procedures. In the third part, you will write Logo programs.

There are 30 possible points, along with 4 extra credit points. The extra credit problems are not much more difficult than the normal problems; we recommend that you complete them all. In addition, participants in the Logo contest can earn up to 3 additional points, along with the glory of victory.

The Logo Language

Before you begin working on the project, review what you have learned in lecture about the Logo language. If you would like to experiment with a working Logo interpreter, try out UCBLogo, which is installed on instructional machines as `logo`.

The following key features of Logo will influence on your interpreter design.

Call Expressions. Logo procedures are called by listing the procedure name, followed by its arguments. Logo call expressions have no parentheses or commas to delimit arguments; only white space separates tokens. The Logo prompt is a question mark.

```
? print 2 2
```

The starter code for your Logo interpreter in `logo.py` can successfully evaluate this simple expression, because it has only one argument, which is a number. The rest of the examples in this section *will not* work until you complete various portions of the project.

Despite their lack of punctuation, call expressions can be nested. That is, the arguments in a call expression can themselves be call expressions. Part of the Logo interpreter's job will be to figure out where each expression begins and ends.

```
? print sum 2 3 5
```

The interpreter identifies the end of a call expression by knowing the number of arguments needed by each procedure. The `print` procedure requires one argument, while the `sum` procedure requires two.

Reading nested Logo expressions can take some practice, and mentally inserting punctuation can aid understanding. For instance, the Logo expression

? print sum product sum 1 2 3 4 13
is equivalent to the Python expression `print(add(mul(add(1, 2), 3), 4))`.

Read-Eval Loop. Unlike Python, the result of evaluating an expression is not automatically printed. Instead, Logo complains if the value of any top-level expression is not `None`.

? 2 You do not say what to do with 2.

In Logo, any top-level expression (i.e., an expression that is not an operand of another expression) must evaluate to `None`. The `print` procedure always outputs `None`, and so printing does not cause an error. Multiple call expressions may appear on the same line of Logo, and the interpreter will evaluate each one. When a top-level expression evaluates to a non-`None` value, the remaining expressions on the line are ignored.

? print 1 2 1 You do not say what to do with 2. ? 1 print 2 You do not say what to do with 1.

Infix Notation. In addition to prefix-notation call expressions, Logo includes seven infix operators: +, -, *, /, =, >, <.

? print 2 + 3 5 ? print (word "re "deliver) = (word "rede "liver) True

Each operator corresponds to a primitive procedure that takes two arguments (e.g., + corresponds to `sum`).

Quotation. Logo has only two built-in data types: words and sentences. Words are like strings without spaces, and also serve as the names of variables. Sentences are like immutable Python lists, which can contain words or other sentences as elements. Logo sentences are also called lists.

Words that represent numbers and boolean values are self-evaluating and are interpreted as word literals. Any token can be interpretered as a word literal if it is preceded (but not followed) by a double quote. Sentence literals are contained in square brackets.

? print "hello hello ? print [hi there] hi there

When a sentence is printed, the delimiting square brackets are omitted so that the result looks more natural. Logo is meant to be conversational.

Words and sentence literals are *quoted* expressions: their contents isn't evaluated. Without the quotation mark, `hello` would be treated as a procedure name!

? print hello I do not know how to hello.

Quoted words serve as arguments to other procedures. Sentences are quoted, in the sense that their contents is not evaluated either. Don't get confused by Logo's syntax -- these quotation marks are not used in pairs; a single one is used before a single word.

? print "hi there hi I do not know how to there.

This example combines several of these concepts, along with the primitive procedures `word` and `sentence`:

? print sentence word "now "here last [the invisible man] nowhere man

Logo must understand that `word` requires two arguments (the quoted words that follow it) while `last` requires one, and that the values returned by `word` and `last` are the two required arguments to `sentence`.

Testing

The file `tests.lg` contains definitions of several Logo procedures that you can examine and test to become more familiar with the language. Each line that prints output is followed by the expected result as a comment. Tests are labeled with the problems to which they correspond.

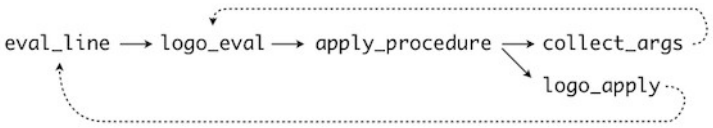
You can run all commands in a file using your Logo interpreter by passing the file name as an argument to `logo.py`.

python3 logo.py tests.lg
You can also compare the output of your interpreter to the expected output by passing the file name to `logo_test.py`.
python3 logo_test.py tests.lg
Don't forget to use the `trace` decorator from the `ucb` module to follow the path of execution in your interpreter.

As you develop your Logo interpreter, you will find that Python raises various uncaught exceptions when evaluating Logo expressions. As a result, your Logo interpreter will crash. By the end of the project, the only exceptions raised should be `LogoError` and `SyntaxError`, which are caught and printed by the Logo read-eval loop.

Part 1: The Evaluator

With your partner, read the first section of `logo.py`, labeled `Evaluator`, and trace the flow of the evaluator. You will find that the call graph below is incomplete in your starter implementation; the dotted lines indicate missing function calls. The `collect_args` implementation does not correctly call `logo_eval` and the `logo_apply` implementation does not correctly call `eval_line`.



The `logo_eval` function evaluates the first well-formed expression in a line of code and returns the result. The argument `line` is a `Buffer` object from `buffer.py`, which contains a list of words and sentences. Make sure that you understand how a `Buffer` class works, as there are many buffers in this project.

Problem 1 (2 pt). The function `collect_args` is supposed to evaluate the next `n` expressions in `line`, and then return their values as a list. However, the recursive call to `logo_eval` is currently

missing. Instead, the provided implementation collects only one argument, and assumes that it is self-evaluating.

Implement `collect_args` correctly. It should make exactly `n` calls to `logo_eval`. After you're finished, you should see the following results:

? sum 2 3 You do not say what to do with the result 5. ? print sum 2 3 5

Your implementation should check for errors in the input line! If there are not `n` arguments available to evaluate, then call `error` to raise a `LogoError`. The error message should include `str(line)` in the result, which shows where in the input line the error occurred.

? sum 2 Found only 1 of 2 args at [sum, 2 >>]

Problem A2 (2 pt). A Logo line can contain more than one call expression, as in the example below. However, the provided implementation of `eval_line` only evaluates a single expression. The correct implementation should evaluate all expressions in a line.

? print 1 print 2 1 2 ? print 1 2 1 You do not say what to do with the result 2. ? 1 print 2 You do not say what to do with the result 1.

Implement `eval_line`, which should take a whole line as input (represented as a `Buffer` instance) and evaluate each expression in turn, but return the first value that is not `None`. The procedure `logo_eval` should still do the work of evaluating the next expression in a line.

Note: `eval_line` does not need to raise or handle errors. The `interpreter_line` and `read_eval_loop` functions provided for you do that.

Problem A3 (2 pt). Logo can contain parentheses in its expressions that indicate where an sub-expression begins and ends. Fill in the `elif` suite in `logo_eval` when `token` is an open parenthesis. For this case, `logo_eval` should call itself, having removed the opening parenthesis, to compute the value of the expression within parentheses. Then, it should verify that a closing parenthesis follows immediately after the next expression. Finally, it should remove that closing parenthesis and return the value of the expression within.

Problem B2 (2 pt). Implement `isquoted` and `text_of_quotation`, which together allow `logo_eval` to handle quoted expressions. The `isquoted` function should return `True` if its argument is either a list or a string that starts with a quotation mark. The function `text_of_quotation` should return its argument stripped of its initial quotation mark if it is a string, or just return its argument otherwise. *Note*: `text_of_quotation` will only be called on values that are quoted.

Problem B3 (2 pt). We need to be able to print the results of Logo computations. Logo provides three primitive procedures for this purpose:

? print [a [b c] d] ; don't show outermost brackets a [b c] d ? show [a [b c] d] ; do show outermost brackets [a [b c] d] ? type [a [b c] d] ; don't start new line after printing a [b c] d?

The `print` and `show` procedures are defined in terms of `type` (done for you in `logo_primitives.py`). Fill in the Python procedure `logo_type`, which uses Python's `print` statement to implmement Logo's `type` procedure. It will take a word or sentence (or

`None`) as input, and print its contents, putting square brackets around any sublists but not around the entire argument.

Applying `logo_type` to a nested list will require a recursive call to `logo_type`. The parameter `top_level` indicates whether the current call is the first one (`True`) or a recursive call (`False`).

Problem 4 (3 pt). Implement `make`, which is Logo's assignment procedure.

Like Python, Logo evaluates call expressions in the context of an environment composed of frames. Familiarize yourself with the `Environment` class in `logo.py`.

Certain primitive procedures need access to the current environment. For example, `make` takes two arguments, a variable name and a value, but the Python procedure that implements it, `logo_make` requires a third argument, the current environment, since the effect of `make` is to modify that environment.

Implementing `make` will require two steps:

1. Modify `apply_procedure` so that the current environment is appended to the `args` list for any `proc` that has a `needs_env` attribute value of `True`.
2. Fill in `Environment.set_variable_value` so that it adds or updates a symbol-to-value binding in the global frame, `self._frames[0]`. *Note*: Always changing the global frame is not quite the correct behavior for Logo's `make` procedure, but you will fix up this implementation later in the project. The `set_variable_value` doctests will not pass until then.

The provided implementation of `lookup_variable` is already sufficient to retrieve values from the global frame.

? make "foo 27 ? print :foo 27

Why the quotation mark before `foo`? Remember, the evaluator would attempt to evaluate the non-existent `foo` procedure otherwise.

Problem 5 (3 pt). The Logo primitives `if` and `ifelse` require as their first arguments the boolean words `True` or `False`. Nothing else is allowed! Predicate procedures must return these boolean words in order to work with `if`. If the first argument to `if` is `True`, the second argument is either evaluated if it is a sentence or returned if not.

? if True [print 3] 3 ? if equalp 3 sum 1 2 [print sum 20 10] 30 ? ifelse lessp 2 1 [print "Yes] [print "No] No ? if 1 [print 3] First argument to "if" is not True or False: 1 ? print if True 5 5 ? print if False 5 None ? ifelse 1 2 3 First argument to "ifelse" is not True or False: 1 ? print ifelse True [sum 1 2] 4 3

Fill in `logo_if` and `logo_ifelse` so that these procedures implement the behavior of Logo's `if` and `ifelse` primitives. Make sure to raise appropriate errors so that the output of your interpreter matches the output above. Call `error(message)` with an appropriate `message` to raise a `LogoError`.

In addition, add doctests to `logo_if` and `logo_ifelse` that test the result of calling `eval_line` on a line that contains a call to `if` or `ifelse`, respectively. Your doctests should

check for correct error messages as well as the correct evaluation of a well-formed line. Make sure that your doctests pass.

Hint: `logo_if` is similar to `logo_run`, in that it evaluates the contents of the list that is passed to it.

Extra Credit 1 (2 pt). Logo is meant to support infix operators as an alternate syntax for call expressions. Rewrite `logo_eval` so that infix expressions are evaluated correctly.

? print 3 + 2 5 ? print 5 + ifelse 2=3 [6-4] [8/5] 6.6
To do so, you will need to *introduce* a helper function `eval_noninfix` that evaluates the first non-infix expression in a line (as `logo_eval` does now). Then, you will need to update the `logo_eval` function with new logic.

Consider evaluating the expression `3 + 2`. Calling `eval_noninfix` will return `3`. Your updated `logo_eval` function must notice that the next token of the line is an infix operator, `+`, find the corresponding procedure, and apply it (using `logo_apply`) to the already-computed value (in this case, `3`) and the value of the first non-infix expression after the infix operator (in this case, `2`). Remember that this following expression might not be a single self-evaluating token; you have to evaluate it.

To summarize, `logo_eval` should:

- Evaluate the first non-infix expression on the line and store its result, for example as `arg0`. (*Hint*: use a call to `eval_noninfix`, a function that you must add.)
- While the next token is an infix symbol:
    - Evaluate the next non-infix expression following the infix symbol (a call to `eval_noninfix`).
    - Apply the procedure for the infix symbol to the two values so far: `arg0` and the value of the following non-infix expression.
    - Update `arg0` to the result of this application.
- Return the most recently computed `arg0`.

The constant dictionary `INFIX_SYMBOLS` has the 7 Logo infix symbols as keys and their corresponding Logo procedure names as values.

Extra Credit 2 (2 pt). Handle operator precedence correctly for expressions that contain multiple infix operators, so that Logo obeys the evaluation rules of standard algebra. Multiplication and division have precedence over addition and subtraction. Moreover, these four arithmetic operators have precedence over the three comparison operators. The `tests.lg` file contains test cases for your implementation.

The three classes of operators are stored in a constant list called `INFIX_GROUPS`.

Precedence can be resolved using your existing design for handling infix operators. Rather than always calling `eval_noninfix`, enforce that the right operand expression of an infix procedure may contain infix expressions of higher (but not lower or the same) precedence, as well as non-infix expressions.

Part 2: Procedures

Here is a Logo procedure definition:

? to factorial :n > if equal? :n 0 [output 1] > output product :n factorial difference :n 1 > end ? print factorial 5 120

A procedure definition spans multiple lines. The procedure name and formal parameters are part of the header, which begins with `to`. The procedure body is entered on lines in response to a continuation prompt, `>`. The body is not evaluated immediately, but instead are stored as part of the procedure text. The special keyword `end` on a line by itself indicates the end of the body.

The `output` procedure is used to specify the return value of a user-defined procedure. Once the `output` procedure is called, the enclosing body is finished; in this example, if the `if` in the first line of the body outputs 1, the second line of the body is not evaluated. The `stop` procedure similarly stops procedure execution, but returns `None`.

Warning: Solutions to problems in this part of the project work together to implement user-defined procedures. Due to the interdependence of these functions, the tests in `tests.lg` for part 2 will not all pass until this whole part is complete. The doctests are designed to give you some early feedback after finishing each question.

Problem 6 (3 pt). Implement `eval_definition`, a function that takes a `Buffer` of tokens as an argument. That buffer contains the procedure name and formal parameter names that follow the keyword `to`. To evaluate a definition:

- Enter an interactive loop in which you read lines of Logo and store them as the body of the procedure. The locally defined function `next_line` will prompt the user and return a parsed line of Logo. This loop ends when the user enters a line that contains only the word `end`.
- Finally, create a `Procedure` Python object and add it to the `env.procedures` dictionary, bound to its name.

Problem 7 (4 pt). Fill in `logo_apply` so that it can apply user-defined procedures. The input `args` is a list of length `n+1`, where `n` is the number of formal parameters for Procedure `proc`, `args[:n]` is the list of evaluated arguments, and `args[n]` is the current environment. There are three important aspects to implementing `logo_apply`.

- **Frames**: The `Environment` object has `push_frame` and `pop_frame` methods. The frame that you push should be a dictionary whose keys are the formal parameter names, and whose values are the arguments to `proc`. That frame should be popped as soon as the procedure application is complete.
- **Lines**: The body `proc.body` is a list of lines. Each line must be placed into a `Buffer`, then evaluated.
- **Results**: The result of evaluating each line should be `None`; Logo should raise an error otherwise (You do not say what to do with the result). The exceptions to this rule are two primitives that can end a procedure invocation early. The procedures `stop` and `output` both return a pair `('OUTPUT', val)`, where `val` is `None` for `stop` and an output value

otherwise. If applying a procedure returns such a pair, then `logo_apply` should return `val`.

Problem 8 (2 pt). Implement `Environment.lookup_variable` to return the value of the first symbol-value binding in the current environment. In a dynamically scoped language, all frames are added to a single environment. The process to look up a variable by name inspects each frame, starting from the most recently added, and returns the first value bound to that name. New frames are appended to the end of `self._frames`.

Problem 9 (2 pt). Modify `Environment.set_variable_value` so that Logo's `make` sets a variable's value in the most recent (innermost) frame in which it was defined, or the global frame, if it wasn't otherwise defined.

Test your work by verifying that all tests in parts 1 and 2 pass when you run:

```
# python3 logo_tests.py tests.lg
```

Your Logo interpreter implementation is now complete.

Part 3: Recursion

Not only is your Logo interpreter itself a tree-recursive program, but it is flexible enough to evaluate *other* recursive programs. Implement the following procedures in Logo at the bottom of `tests.lg`.

Problem A10 (2 pt) Implement the `filter` procedure, which takes two arguments, a procedure name and a sentence. It outputs a sentence that contains all elements of the input sentence for which applying the named procedure outputs `True`. *Hint*: use the `fput` primitive from `logo_primitives.py` to extend an existing list into a new list. The provided `apply_1` procedure may be useful.

Problem A11 (2 pt). Implement the `count_change` procedure, which counts all of the ways to make change for a `total` amount, using coins with various denominations (`denoms`), but never uses more than `max_coins` in total. Write your implementation in `tests.lg`. The procedure definition line is provided, along with U.S. denominations.

Problem B10 (2 pt) Implement the `reduce` procedure, which takes three arguments, a procedure name, a sentence, and a starting value. It outputs a value that results from repeatedly applying the named procedure to the accumulated value and a subsequent element of the input sentence. *Hint*: The provided `apply_2` procedure may be useful.

Problem B11 (2 pt). Implement the `count_partitions` procedure, which counts all the ways to partition a positive integer `total` using only pieces less than or equal to another positive integer `max_value`. The number `5` has 5 partitions using pieces up to a `max_value` of `3`:

3, 2 (two pieces) 3, 1, 1 (three pieces) 2, 2, 1 (three pieces) 2, 1, 1, 1 (four pieces) 1, 1, 1, 1, 1 (five pieces)

Problem 12 (3 pt). Implement the `list_partitions` procedure, which lists all of the ways to partition a positive integer `total` into at most `max_pieces` pieces that are all less than or equal

to a positive integer `max_value`. *Hint*: Define a helper function to construct partitions. The provided `len` procedure may be useful.

Congratulations! You have finished the final project for 61A. You are not only a rock star, but a proper computer scientist!

Contest: Recursive Art

Logo has a number of primitive drawing procedures that are collectively called "turtle graphics". The *turtle* represents the state of the drawing module, which has a position, an orientation, a pen state (up or down), and a pen color. The `load_turtle_graphics` function in `logo_primitives.py` lists these procedures and their implementations. The Python documentation of the turtle module contains more detail.

Logo Contest (3 pt). Create a visualization of an iterative or recursive process of your choosing, using turtle graphics. Your implementation must be written entirely in Logo, using the interpreter you have built (no fair extending the interpreter to do your work in Python, but you can expose other turtle graphics functions from Python if you wish).

Prizes will be awarded for the winning entry in each of the following categories.

- **Featherweight.** At most 128 words of Logo, not including comments and delimiters.
- **Heavyweight.** At most 1024 words of Logo, not including comments and delimiters.

Entries (code and results) will be posted online, and winners will be selected by popular vote. The voting instructions will read:

> Please vote for your favorite entry in this semester's 61A Recursion Exposition contest. The winner should exemplify the principles of elegance, beauty, and abstraction that are prized in the Berkeley computer science curriculum. As an academic community, we should strive to recognize and reward merit and achievement (translation: please don't just vote for your friends).

To improve your chance of success, you are welcome to include a title and descriptive haiku in the comments of your entry, which will be included in the voting. Place your completed entry into the `contest.lg` file.