CS61A Notes – Week 13: Interpreters

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.

Evaluator

Logo is evaluated one line at a time. The sentence returned from `parse_line` is passed to the `eval_line` function, which evaluates each expression in the line. The `eval_line` function repeatedly calls `logo_eval`, which evaluates the next full expression in the line until the line has been evaluated completely, then returns the last value. The `logo_eval` function evaluates a single expression.

The form of a multi-element expression in Logo can be determined by inspecting its first element. Each form of expression has its own evaluation rule.

1. A primitive expression (a word that can be interpreted as a number, True, or False) evaluates to itself.
2. A variable (a string that start with ":") is looked up in the environment.
3. A definition (which starts with "to") is handled as a special case.
4. A quoted expression evaluates to the text of the quotation, which is a string without the preceding quote. Sentences (represented as Python lists) are also considered to be quoted; they evaluate to themselves.
5. A call expression looks up the operator name in the current environment and applies the procedure that is bound to that name.

1. Fill in the code for logo_eval

```
def logo_eval(line, env):
    token = line.pop()
    if isprimitive(token):                    #check if primitive
        return token
    elif isvariable(token):                   #check if variable
        return env.lookup_variable(variable_name(token))
    elif isdefinition(token):                 #check if definition
        return eval_definition(line, env)
    else:
        procedure = env.procedures.get(token, None)
        if not procedure:
            error('I do not know how to {0}.'.format(token))
        return apply_procedure(procedure, line, env)
```

2. How many calls to `eval` and `apply` occur when the following lines are evaluated?

? sum 1 2
3

? sum word 1 2 3
5

More Dynamic Scope

Recall that Python uses Lexical Scoping, which controls what frame a newly created frame points to. Lexical Scoping says: when calling a function, create a new frame that extends the environment that the function was defined in. Logo, however, uses a different rule - it uses Dynamic Scoping. Dynamic Scoping says: when calling a function, create a new frame that extends the current frame.

```
? make "x 3
? to scope :x
   helper 5
 end
? to helper :y
   print (sentence :x :y)
 end
? scope 4
; expects 4 5
```

Under lexical scope, this would return something along the lines of `[3 5]`. However, with dynamic scope, helper has access to the `:x` variable as it extends the frame with that variable. Thus, we end up with a sentence of `4 5` printed to the screen.

```
? to bar :n
   output sentence :n baz :n+1
 end
? to baz :m
   output sentence :n :m
 end
? make "n 7
? bar 10
```

1. What is the result of this using lexical scope? Dynamic scope?
   lexical: [10 7 11]
   dynamic: [10 10 11]

2. Draw a corresponding environment diagram for each scope.

Assignment

Logo supports binding names to values. As in Python, a Logo environment consists of a sequence of frames, and each frame can have at most one value bound to a given name. In Logo, names are bound with the make procedure, which takes as arguments a name and a value:

```
? make "x 2
```

The values bound to names are retrieved by evaluating expressions that begin with a colon:

```
? print :x
2
```

Assignment rules:

1. If the name is already bound, `make` re-binds that name in the first frame in which the name is bound (automatic non-local Python assignment)
2. If the name is not bound, `make` binds the name in the global frame

Given:

```
? to foo :n
   make "n 6
   make "m 7
   print bar :n
   print :n
 end
? to bar :m
   make "n 4
   make "m 8
   output :m
 end
```

1. What is printed when we call `foo 10`?

8
4

2. What is printed when we call `print :n`? How about `print :m`?

n has no value
7

Procedures

Logo supports user-defined procedures using definitions that begin with the "`to`" keyword. The first line of a definition gives the name of the new procedure, followed by the formal parameters as variables. The lines that follow constitute the body of the procedure, which can span multiple lines and must end with a line that contains only the token "`end`".

Logo's application process for a user-defined procedure is similar to the process in Python. Applying a procedure to a sequence of arguments begins by extending an environment with a new frame, binding the formal parameters of the procedure to the argument values, and then evaluating the lines of the body of the procedure in the environment that starts with that new frame.

```
class Procedure(object):
   """ A Logo procedure, either primitive or user-defined

   name: The name of the procedure.  For primitive procedures with multiple
         names, only one is stored here.

   arg_count: Number of arguments required by the procedure.

   body: A Logo procedure body is either:
           a Python function, if isprimitive == True
           a list of lines,   if isprimitive == False

   isprimitive: whether the procedure is primitive.
   """

   def __init__(self, name, arg_count, body, isprimitive=False, ...):
       self.name = name
       self.arg_count = arg_count
       self.body = body
       self.isprimitive = isprimitive
       ...
```

Write the primitives section of `logo_apply`. Remember that it might not always be the case that the procedure will work on a given argument, so we should use some exception handling here.

```
def logo_apply(proc, args):
   if proc.isprimitive:
         try:
             return proc.body(*args)
         except Exception as e:
             error(e)

   else:
      ...                #You do not have to write this part
```

Environments

First, let's review how lookup works for variables in environments:

1. Look for binding for variable of that name in the current frame.
2. If binding exists, return the value.

3. If not, look in your enclosing frame.
4. If you reach the global environment without finding a value, raise an error.

How are we going to implement this in an interpreter? We first have to understand how to represent environments and frames inside our program.

The concept of a frame is simple - a frame tells us variables and their values. Thus, we can think of a frame as a set of bindings. We will implement a frame as a dictionary, whose keys are variables names and whose values are the variables' corresponding values in that frame. For example, consider the following:

```
? to qux :a :b :c
   output (sum :a :b :c)
  end
? qux 1 2 3
```

When we call `qux 1 2 3`, the frame generated by that call can be represented as `{'a' : 1, 'b' : 2, 'c' : 3}`.

Now that we've defined frames, it's intuitive to define an environment as a list of frames. The list is ordered, and we denote one end of the list to be the current frame, and the other end to be the global frame. Each frame is adjacent to the frame that points to it, and the frame that it points to.

Suppose we made these calls in logo:

```
? to garply :m
   output foo :m*2
  end
? to foo :m
   output bar :m/2
  end
? to bar :n
   output 5
  end
? show garply 4
```

1. Just before `garply` finishes outputting its result, what does the list of frames look like?

[{'m' : 4}, {'m' : 8}, {'n' : 4}]

2. Finish the definition of the Environment class:

```
class Environment(object):
   """An environment holds procedure (global) and name bindings in frames."""
   def __init__(self, get_continuation_line=None):
```

```
        self.get_continuation_line = get_continuation_line
        self.procedures = load_primitives()
        self._frames = [dict()] # The first frame is a list of the global frame

    def push_frame(self, frame):
        """Add a new frame, which contains new bindings."""
        self._frames.append(frame)

    def pop_frame(self):
        """Discard the last frame."""
        self._frames.pop()
```