Interpreters 11

Computer Science 61A

July 28, 2016

1 Calculator

We are beginning to dive into the realm of interpreting computer programs – that is, writing programs that understand other programs. In order to do so, we'll have to examine programming languages in-depth. The *Calculator* language, a subset of Scheme, was the first of these examples. In today's discussion, we'll be extending Calculator with variables and user-defined functions.

The Calculator language is a Scheme-syntax language that currently includes only the four basic arithmetic operations: $+$, $-$, $*$, and $/$. These operations can be nested and can take varying numbers of arguments. Here's a few examples of Calculator in action:

```
calc> (+ 2 2)
4

calc> (- 5)
-5

calc> (* (+ 1 2) (+ 2 3))
15
```

Our goal now is to write an interpreter for this language, and extend its functionality to variables and user-defined functions. The job of an interpreter is to evaluate expressions. So, let's talk about expressions.

A Calculator expression is just like a Scheme list. To represent Scheme lists in Python, we use `Pair` objects. For example, the list `(+ 1 2)` is represented as `Pair('+', Pair(1,` `Pair(2, nil)))`. The `Pair` class is similar to the Scheme procedure `cons`, which would represent the same list as `(cons '+ (cons 1 (cons 2 nil)))`.

`Pair` is very similar to `Link`, the class we developed for representing linked lists. In addition to `Pair` objects, we include a `nil` object to represent the empty list. `Pair` instances have methods:

1. `__len__`, which returns the length of the list.
2. `__getitem__`, which allows indexing into the pair.
3. `map`, which applies a function, `fn`, to all of the elements in the list.

`nil` has the methods `__len__`, `__getitem__`, and `map`.

Here's an implementation of what we described:

```
class nil:
    """Represents the special empty pair nil in Scheme."""
    def __repr__(self):
        return 'nil'

    def __str__(self):
        return '()'

    def __len__(self):
        return 0

    def __getitem__(self, i):
        raise IndexError('Index out of range')

    def map(self, fn):
        return nil

nil = nil() # this hides the nil class *forever*

class Pair:
    """Represents the built-in pair data structure in Scheme."""
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):
        return 'Pair({}, {})'.format(repr(self.first), repr(self.
           second))
```

```
    def __str__(self):
        result = '(' + str(self.first)
        while isinstance(self.second, Pair):
            self = self.second
            result += ' ' + str(self.first)
        if self.second is nil:
            return result + ')'
        return result + ' . ' + str(self.second) + ')'

    def __len__(self):
        return 1 + len(self.second)

    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.second[i-1]

    def map(self, fn):
        return Pair(fn(self.first), self.second.map(fn))
```

1.1 Questions

1. Translate the following Calculator expressions into calls to the `Pair` constructor.

   `> (+ 1 2 (- 3 4))`

   Solution:
   ```
   >>> Pair('+', Pair(1, Pair(2, Pair(
           Pair('-', Pair(3, Pair(4, nil))), nil))))
   ```

   `> (+ 1 (* 2 3) 4)`

   Solution:
   ```
   >>> Pair('+', Pair(1, Pair(Pair( '*', Pair(2, Pair(3, nil))
     ),
           Pair(4,nil))))
   ```

2. Translate the following Python representations of Calculator expressions into the proper Scheme syntax:

   `>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))`

Solution:

```
> (+ 1 2 3 4)
```

```
>>> Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

Solution:

```
> (+ 1 (* 2 3))
```

2 Evaluation

Evaluation discovers the form of an expression and executes a corresponding evaluation rule.

We'll go over two such expressions now:

1. *Primitive* expressions are evaluated directly. For example, the numbers 3.14 and 165 just evaluate to themselves, and the string "+" evaluates to the `calc_add` function.
2. *Call* expressions are evaluated in the same way you've been doing them all semester:
   (1) **Evaluate** the operator.
   (2) **Evaluate** the operands from left to right.
   (3) **Apply** the operator to the operands.

Here's `calc_eval`:

```
def calc_eval(exp):
    """Evaluates a Calculator expression represented as a Pair.
       """
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first),
                          list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:  # Just a number
        return exp
```

And here's `calc_apply`:

```
def calc_apply(op, args):
    """Applies an operator to a Pair of arguments."""
```

```
    return op(*args)
```

2.1 Questions

1. Suppose we typed each of the following expressions into the Calculator interpreter. How many calls to `calc_eval` would they each generate? How many calls to `calc_apply`?
   `> (+ 2 4 6 8)`

   Solution:
   6 calls to **eval**. 1 call to **apply**.

   `> (+ 2 (* 4 (- 6 8)))`

   Solution:
   10 calls to **eval**. 3 calls to **apply**.

3 Defining Variables and Functions

Let's extend the functionality of our Calculator interpreter to allow us to define new variables and functions. We want this to work the same way as how we would bind a symbol to a value in Scheme, e.g.: `(define x (+ 3 4))`.

Using our current `calc_eval` implementation, we would map `calc_eval` on each argument in `exp.second`. However, in the example above, we should not call `calc_eval` on `x`. Hence, we introduce special forms.

First, we will implement `do_define_form`, which will allow us to bind symbols to values. Then, we will implement `do_lambda_form`, which will allow us to create user-defined functions.

3.1 Questions

1. Before we can create functions and bind symbols to values, we need a way to keep track of different frames and environments. Fill in the `define` and `lookup` methods in the `Frame` class. The `define` method should assign the key `name` to the value `value` in the bindings of the current frame. The `lookup` method should return the value bound to `name` in the current frame, or lookup in the parent if there is one. Otherwise, raise a `NameError`.

```
class Frame:
    def __init__(self, parent=None):
        self.bindings = {}
        self.parent = parent

    def define(self, name, value):
```

Solution:

```
        self.bindings[name] = value
```

```
    def lookup(self, name):
```

Solution:

```
        if name in self.bindings:
            return self.bindings[name]
        elif self.parent is None:
            raise NameError('name {} is not defined'.format
               (name))
        else:
            return self.parent.lookup(name)
```

```
global_frame = Frame()
```

Note that, to handle environments and the `define` and `lambda` special forms, we have to modify `calc_eval` as follows:

```
def calc_eval(exp, env):
    """Evaluates a Calculator expression."""
    if isinstance(exp, Pair):
        first, second = exp.first, exp.second
        if first in SPECIAL_FORMS:
            return SPECIAL_FORMS[first](second, env)
        op = calc_eval(first, env)
        args = second.map(lambda exp: calc_eval(exp, env))
        return calc_apply(op, list(args))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    elif isinstance(exp, str):
        return env.lookup(exp)
    else:
        return exp
```

`calc_eval` has to take in an additional parameter `env`, which is the current frame that `exp` is being evaluated in. If `exp` is a string, meaning it is a symbol, we simply look it up in `env`.

To handle special forms, we create a dictionary `SPECIAL_FORMS` that maps the strings `"define"` and `"lambda"` to the functions `do_define_form` and `do_lambda_form`, respectively. These functions take in the rest of `exp` and perform the order of evaluation specific to their special form.

```
SPECIAL_FORMS = {'define': do_define_form,
                 'lambda': do_lambda_form}
```

2. Let's now implement `do_define_form`. This function takes in `exp`, which is the rest of the expression after the `define`), and a frame `env` and binds the name given by `exp.first` to the value that `exp.second.first` evaluates to in `env`.

```
def do_define_form(exp, env):
    target = exp.first
```

Solution:

```
    value = calc_eval(exp.second.first, env)
    env.define(target, value)
```

```
    return target
```

3. We can now bind symbols to values! But there's more work to be done to allow for user-defined functions.

   We will first implement a class that represents procedures. Instances of the `LambdaProcedure` class are created with `formals`, a `Pair` containing the names of the parameters, `body`, a `Pair` representing the expression that is the body of the procedure, and `env`, the frame where the procedure was created.

   We will restrict ourselves to procedures with only one expression in the body, similar to how lambda functions are restricted in Python. In the project, you will have to handle arbitrary procedures.

   Implement the `make_call_frame` method, which takes in a `Pair` of arguments `args` and creates and returns a new frame where the formal parameters of the procedure are bound to the elements of `args`. Make sure the frame that is created has the correct parent.

```
class LambdaProcedure:
    """A procedure defined by a lambda expression."""

    def __init__(self, formals, body, env):
        self.formals = formals
        self.body = body
        self.env = env

    def make_call_frame(self, args):
```

Solution:

```
        frame = Frame(self.env)
        for i in range(len(self.formals)):
            frame.define(self.formals[i], args[i])
        return frame
```

Now, it is easy to implement `do_lambda_form`:

```
def do_lambda_form(exp, env):
    return LambdaProcedure(exp.first, exp.second.first, env)
```

Evaluating a `lambda` special form simply creates and evaluates to the procedure itself. However, there is one more step: currently, our `calc_apply` does not know how to apply user-defined functions. Modify it below so that it can handle instances of the `LambdaProcedure` class:

```
def calc_apply(op, args):
    """Applies an operator to a Pair of arguments."""
    if isinstance(op, LambdaProcedure):
```

Solution:

```
        new_env = op.make_call_frame(args)
        return calc_eval(op.body, new_env)
```

```
    else:
        return op(*args)
```

We have now added variables and procedures to our Calculator language! With a few small changes to the parser (to allow for symbols) and the REPL (to handle environments), we are able to use our interpreter like this:

```
calc> (+ 4 5)
9
calc> (define x 4)
x
calc> (+ x 6)
10
calc> ((lambda (x) (* x x)) 7)
49
calc> (define f (lambda (x y) (* (+ x y) (- x y))))
f
calc> (f 5 4)
9
```

Download the Calculator interpreter from Lecture 21 and see if you can figure out the necessary changes to get this to work!