CS61A Week 11 Notes

Interpretation

To help our understanding of interpreters, let's step through a few critical components of the project. We will carefully examine a few classes in the framework you are given for the project.

REPL Loop

The REPL loop is the loop that drives the interpreter. We have already discussed it in abstract. Let's look at a concrete example.

```
def read_eval_print(prompt = None):
    while True:
        try:
            if prompt is not None:
                print(prompt, end = "")
            sys.stdout.flush()
            expr = scm_read()
            if expr is THE_EOF_OBJECT: #end-of-file
                return
            val = scm_eval(expr)
            if prompt is not None and val is not UNSPEC:
                scm_write(val)
                scm_newline()
        except SchemeError as exc:
            if not exc.args[0]:
                print("Error", file=sys.stderr)
            else:
                print("Error: {0}".\
                      format(exc.args[0]), file=sys.stderr)
            sys.stderr.flush()
```

Questions:

1. Can you spot each of R-E-P-L ?
2. What ends the interpreter? EOF expression
3. What values are not printed? None and UNSPEC

Frames

Our first focus is the EnvironFrame class. This represents a single frame in the environment diagram.The frames behave like “linked dictionaries” - each frame acts like a dictionary, and contains a pointer to its enclosing frame.

```
class EnvironFrame:

    def __init__(self, enclosing):
        """An empty frame that is attached to the frame ENCLOSING."""
        self.inner = {}
        self.enclosing = enclosing

    def __getitem__(self, sym):
        return self.find(sym).inner[sym]

    def __setitem__(self, sym, val):
        self.find(sym).inner[sym] = val

    ...

    def find(self, sym):
        """The environment frame at or enclosing SELF that
        defined SYM.  It is an error if it does not exist."""
        e = self
        while e is not None:
            if sym in e.inner:
                return e
            e = e.enclosing
        raise SchemeError("unknown identifier: {0}".format(str(sym)))

     def make_call_frame(self, formals, vals):
       return EnvironFrame(self) # replace w/ your solution



   def define(self, sym, val):
        """Define Scheme symbol SYM to have value VAL in SELF."""
        self.inner[sym] = val
```

Questions

1. You have a frame F. How do you get the value of a variable x? F[‘x]
2. How do you create a new variable y with value 3? F.define(‘y’, 3)
3. How do you tell if a frame is the global frame? f.enclosing = None
4. Look at the make_call_frame function signature. It is used to create a new frame for a function call. FORMALS a set of formal parameters, which is either an ordinary Scheme

list or the last cdr of FORMALS is a symbol, in which case the symbol should point to values in rest of vals. (Look at the attached definition of SchemeValue)

a. How do we check if formals is a symbol? If it is the empty list? F.symbolp(), F.nullp().
b. How do we create the new frame? What is its enclosing frame? EnvironFrame(self)

Review

The third (no more until the final, I swear) CS61A midterm is next Wednesday! Remember that the test will cover "the entirety of human knowledge from the beginning of recorded history with particular emphasis on the material presented in this course." Let's think back to some (NOT ALL) topics we've seen in the course so far:

- Simple Python Functions
- Environment Diagrams
- Higher Order Functions
- Sequences (e.g. Strings and Tuples)
- Nonlocal values
- Common Data Structures (e.g. Python lists, recursive lists, trees, dictionaries, and sets)
- Recursion (including tree recursion, mutual recursion, and tail-recursive functions)
- Object Oriented Programming
- Orders of Growth
- Interpretation

KEEP IN MIND THAT THIS LIST IS NOT COMPLETE.

Questions

Given the following class definition for `PyTree` (a type of tree) answer the following questions:

```
class PyTree:
    """A Tree for Python"""
    def __init__(self, item, *children):
        self._item = item
        self._children = children

    @property
    def children(self):
        return self._children

    @property
    def item(self):
        return self._item

    @property
    def is_leaf(self):
```

```
        return self._children is None

    def __str__(self):
        return "{0}{1}".format(self._item, self_children)
```

1. Write the function `pytree_map` which takes a `PyTree` and a function and returns a new `PyTree` where each element is the result of taking the corresponding element in the original `PyTree` and applying the function to it.

```
def pytree_map(fn, ptree):
    """Map for PyTrees
    >>> test_tree = PyTree(1,
    ...                    PyTree(3,
    ...                           PyTree(43),
    ...                           PyTree(27)),
    ...                    PyTree(5,
    ...                           PyTree(22)))
    ...
    >>> str(test_tree)
    1(3(43, 27), 5(22))
    >>> str(pytree_map(lambda x: x + 1, test_tree))
    2(4(44, 28), 6(23))
    >>> str(pytree_map(lambda x: x * x, test_tree))
    1(9(1849, 729), 25(484))
    """
    return PyTree(fn(ptree.item),
                  *[ptree_map(fn, child) for child in ptree.children])
```

2. Say I wanted to make the constructor require that all children of a PyTree are also PyTrees. Write a revised __init__ method for PyTree which raises a ValueError if one of the children is not a PyTree.

```
def __init__(self, item, *children):
    """Raises a ValueError if there is a child which is not a
    PyTree.
    >>> good_tree = PyTree(1, PyTree(2))
    >>> bad_tree = PyTree(1, 2)
    Traceback (most recent call last):
        ...
    ValueError: non-PyTree child!
    """
    """YOUR CODE HERE"""
    self._item = item
    for child in children:
        if type(child) is not PyTree:
            raise ValueError("non-PyTree child!")
    self._children = children
```

3. What is the order of growth of the following functions?

```
def bar(n):   0(1)
    if n % 7 == 0:
        return "Bzzst"
    else:
        return bar(n - 1)
def garply(n):  0(log(n))
    if n <= 1:
        return 1
    else:
        return 3 * garply(n / 3) + 1
```

```
def foo(n):     0(n)
    if n <= 1:
        return 1
    elif n % 2 == 1:
        return 1 + foo((n - 1) / 2) + foo((n - 1) / 2)
    else:
        return foo(n / 2) + foo(n / 2)
```

4. Draw the environment diagram for the following code.

```
def a(x=5):
    x += 2
    def foo():
        global y
        y += 2
    def bar():
        while y < 4:
           foo()
        return x
    return bar
y = 0
garply = a()
garply()
baz = a()
garply()
baz()
```

5. Say we wanted to write a data abstraction for email messages. An email has a sender, a recipient, and the body. Fill in the constructor below so that it works with the given selectors:

```
def email_sender(eml):
    return eml('sender')

def email_recipient(eml):
    return eml('recipient')

def email_message(eml):
    return eml('msg')

def set_email_sender(eml, s):
    eml('set-sender', s)

def set_email_recipient(eml, r):
    eml('set-recipient', r)

def set_email_message(eml, m):
    eml('set-message', m)

def make_email(sender, recipient, message):
    """YOUR CODE HERE"""
    def email(action, *args):
        nonlocal sender, recipient, message
        if action == "sender":
            return sender
        elif action == "set-sender":
            sender = args[0]
        elif action == "recipient":
            return recipient
        elif action == "set-recipient":
            recipient = args[0]
        elif action == "message":
            return message
        elif action == "set-message":
            message = args[0]
        else:
            print("Sorry, don't know that one!")
    return email
```

6. Say I wanted to memoize a function but I had a very small amount of memory. In order to get around this, I decide to memoize only the 100 most recent *different* calls to the function. Complete the following definition for low_memory_memoize which takes a function and returns a new function which does the same thing except that it memoizes the 100 most recent calls.

```
def low_memory_memoize(fn):
    """YOUR CODE HERE"""
    cache = []
    def item_in_cache(item):
        for key, value in cache:
            if key == item:
                return value
        return False
    def memoized_fn(*args):
        """YOUR CODE HERE"""
        nonlocal cache
        if item_in_cache(args):
           return item_in_cache(args)
        else:
           res = fn(*args)
           if len(cache) == 100:
              cache = cache[1:]
           cache.append((args, res))
           return res
    return memoized_fn
```