SCHEME INTERPRETER GUIDE 4

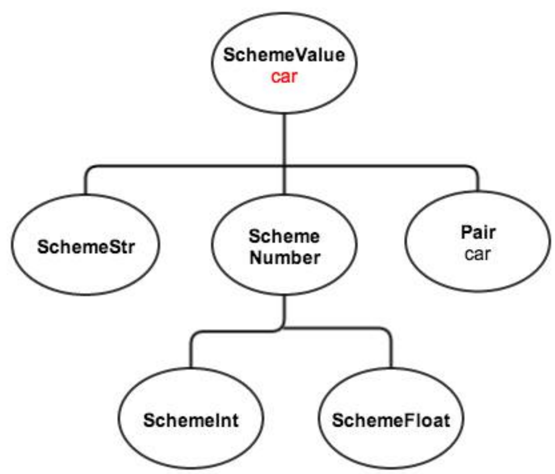
COMPUTER SCIENCE 61A

July 28, 2014

1 Scheme Values

Back in Python, we had all these objects (i.e. lists, tuples, strings, integers) which inherited from the superclass *object*. Since our interpreter is trying to interpret Scheme, we've decided to represent our inputs as Scheme objects instead.

The Scheme objects implemented are `Pairs` (lists), `Procedures` (functions), `SchemeSymbol`, `SchemeStr`, etc and all of them inherit from the superclass `SchemeValue`. The following diagram showcases some of the relationship.



For example, when we parse our input `stk> (car '(1 2))`, we parse it to

```
Pair(SchemeSymbol(car), Pair(Pair(SchemeInt(1), \
  Pair(SchemeInt(2), nil)), nil)).
```

When we evaluate this parsing, we eventually call the `car` method on `Pair(SchemeInt(1), Pair(SchemeInt(2), nil))`. We give `Pairs` a `car` method.

However, the user might input bad commands like `stk> (car 1)`. We try to call the `car` method on a `SchemeInt`; however, because there is no `car` method in `SchemeInt`, Python would throw an `AttributeError: SchemeInt has no attribute car` and our Scheme interpreter would crash. Our program is not allowed to crash. What we can can do is implement a `car` method into `SchemeInt` and have that `car` method raise a `SchemeError`. Well build our interpreter to catch `SchemeErrors` and deal with them properly. (Remember that we must distinguish between errors from user input and errors from our program, if it is faulty.)

However, what if the user tries to input `stk> (car 'hi)` or `stk> (car 1.1)`? Now our program will crash because there is no `car` method in `SchemeStr` or `SchemeFloat`. How do we deal with the other Scheme objects not mentioned? Well we could try putting in a car method in all those Scheme objects, but thats repetitive. What if we only put it in `SchemeValue` instead? Then if we try to call the `car` method on a `SchemeInt` or `SchemeStr`, Python will take advantage of inheritance to get the `car` method from `SchemeValue`.

To summarize, `SchemeValue` was created to contain all the methods that would be called on the implemented Scheme object instances. Most of the methods in `SchemeValue` will raise a `SchemeError` because they only are valid on one type of Scheme object. The other methods in `SchemeValue` are the default definitions for most Scheme objects (like `booleanp`, which returns `True` because it checks if the object is a boolean or not.) An instance object of `SchemeValue` will never be created. The various subclasses of `SchemeValue` will be implemented and each will override certain methods of `SchemeValue` when needed.

2 Parsing Scheme

When reading the previous guide, you may have wondered when we convert the operator and operands to Scheme expressions? The answer is in the parsing stage.

This is because when we get to the eval stage, everything has to be in Scheme expression form in order to be evaluated. Therefore, in the steps before eval, we have to change the `2` that is inputted into a `SchemeInt(2)`. We tackle this when we parse.

The parsing we do in our Scheme Interpreter follows the same general idea as the one that you did for your Calculator homework. This is because Scheme uses *prefix* notation, just like the Calculator did.

Below is the code from homework, compared with the code given for the Scheme project. The `read_exp` and `read_until_close` function from homework turned the tokenized line into a combination of `Pairs` once they encountered a expression with parentheses around it. The `scheme_read` and `read_tail` from the project do something analogous to that.

Right now, we want to focus on the part where we convert a single token into a Scheme object. We see that we do that in the beginning `if` conditions for `scheme_read`. Where could the homework equivalent of that go? Which function did we create to focus on a particular token at a time? Numberize.

To compare the parts of the code that perform the same function, we've color coded corresponding section of the code in the side-by-side comparisons. Can you see how we can replace the `try-except` statement in `numberize` with the `if` statements in `scheme_read`?

```
def scheme_read(src):
    if src.current() is None:
        raise EOFError
    val = src.pop()
    if type(val) is int or \
      type(val) is float:
        return scnum(val)
    elif type(val) is bool:
        return scbool(val)
    elif val not in DELIMITERS:
        if val[0] == '"':
            return scstr(eval(val))
        else:
            return intern(val)
    elif val == "'":
        "*** YOUR CODE HERE ***"
    elif val == "(":
        return read_tail(src)
    else:
        raise SyntaxError("unexpected \
          token: {0}".format(val))

def read_tail(src):
    try:
        if src.current() is None:
            raise SyntaxError("unexpected \
              end of file")
        if src.current() == ")":
            src.pop()
            return nil
        "*** YOUR CODE HERE ***"
        first = scheme_read(src)
        rest = read_tail(src)
        return Pair(first, rest)
    except EOFError:
        raise SyntaxError("unexpected \
          end of file")
```

```
def numberize(atomic_exp):
    try:
        return int(atomic_exp)
    except ValueError:
        try:
            return float(atomic_exp)
        except ValueError:
            return atomic_exp

def read_exp(tokens):
    if tokens == []:
        raise SyntaxError('unexpected \
          end of input')
    token, rest = tokens[0], tokens[1:]
    if token == ')':
        raise SyntaxError('unexpected )')
    elif token == '(':
        if rest == []:
            raise SyntaxError(' \
              mismatched parentheses')
        elif rest[0] == ')':
            raise SyntaxError('empty \
              combination')
        return read_until_close(rest)
    else:
        return numberize(token), rest

def read_until_close(tokens):
    if tokens == []:
        raise SyntaxError('unexpected \
          end of input')
    token, rest = tokens[0], tokens[1:]
    if token == ')':
        raise SyntaxError('unexpected )')
    elif token == '(':
        if rest == []:
            raise SyntaxError(' \
              mismatched parentheses')
        elif rest[0] == ')':
            raise SyntaxError('empty \
              combination')
```

It seems that in `scheme_read`, instead of having `SchemeSymbol`, `SchemeStr`, etc, we have `intern`, `scstr`, etc. The latter are functions that we wrote to create their analogous Scheme objects. (If you look at the code for these functions in *scheme_primitives.py*, you'll understand why we used these functions instead of directly instantiating the class.)

The other main type of Scheme object that hasn't been mentioned yet is `Procedures`. `Procedures` are our way of implementing functions. We'll discuss more of that in the next section.

To summarize the relationship between Scheme data types to their respective Python code, we've created this handy table:

Scheme Data Type	Our Internal Representation Class	Python Code
	Types defined in *scheme_primitives.py*	
Numbers: 0, 3.2	`SchemeInt` and `SchemeFloat`	`scnum(0),scnum(3.2)`
Symbols: merge, x	`SchemeSymbol`	`intern('merge'),intern('x')`
Strings: "foo"	`SchemeStr`	`scstr('foo')`
Booleans: #t, #f	`scheme_true` and `scheme_false`	`scheme_true, scheme_false`
Pairs: (a . b)	`Pair`	`Pair(intern('a'), intern('b'))`
nil: ()	`nil`	`nil`
Lists: (a b)	`Pair` and `nil`	`Pair(intern('a'), Pair(intern('b'), nil))`
okay	`okay`	`okay`
	Types defined in *scheme.py*	
Functions	`PrimitiveProcedure`, `LambdaProcedure`, `MuProcedure`	`PrimitiveProcedure(...)`, `LambdaProcedure(...)`, `MuProcedure(...)`

3 Functions in Scheme

There are two types of functions: Procedures and Special Forms. Lets talk about Procedures first.

There are two types of procedures: Built-In and User-Defined. User-Defined Procedures are functions that we define. So in our Scheme interpreter, it'll be `lambda`s and `mu`s (`mu`s are `lambda`s but with a different type of lookup process). Because you'll be implementing user-defined procedures for the project, we'll just talk about the built-in procedure.

3.1 Built-In Procedures

Let's go back to the example in the beginning: `stk> (car '(1 2))`. This gets parsed to:

```
Pair(SchemeSymbol(car), Pair(Pair(SchemeInt(1), \
  Pair(SchemeInt(2), nil)), nil)).
```

When we evaluate this expression, we evaluate each subexpression: `SchemeSymbol(car)` and `Pair(SchemeInt(1), Pair(SchemeInt(2), nil))`. When `SchemeSymbol(car)` gets evaluated in `scheme_eval`, we see that it is a `SchemeSymbol` and then lookup its value in the `Frames` that we create in our interpreter. The value that we associate with `SchemeSymbol(car)` is `Procedure(scheme_car)`.

```
def scheme_eval(expr, env):
    ...
    while env is not None:
        if expr is None:
            raise SchemeError("Cannot evaluate an undefined \
              expression.")
        if scheme_symbolp(expr):
            expr, env = env.lookup(expr).get_actual_value(), None
    ...
```

Procedures are created by taking in a function. In our example, `scheme_car` is a function that we created. `scheme_car` takes in a variable and calls the `car` method of that variable. When we finish evaluating the subexpressions, we then apply the operator to the operands with `scheme_apply(Procedure(scheme_car), Pair(SchemeInt(1), Pair(SchemeInt(2), nil)))`.

The `apply` method of Procedure will call `scheme_car` with the operand and our interpreter will get back `SchemeInt(1)`.

```
def scheme_eval(expr):
    ...
    procedure = scheme_eval(first, env)
    args = procedure.evaluate_arguments(rest, env)
    if proper_tail_recursion:
        "*** YOUR CODE HERE ***"
    else:
        expr, env = scheme_apply(procedure, args, env), None
    ...
```

3.2 Special Forms

Special Forms are different from Procedures in that Procedures will evaluate all of its operands while a Special Form will only evaluate some. Some examples of Special Forms include the functions `or`, `and`, `let`. Where must we handle these special forms? In `scheme_eval` because if it was in `scheme_apply`, then all of the operands will have already been evaluated, which we don't want.

Since each special form is evaluated differently, each special form will have its own function. For example, if we get a `or` then we'll call `do_or_form`. If we get an `and`, then we'll call `do_and_form`. Since we have about 10 special forms, we don't want to have a series of `if...else...` statements inside of our `scheme_eval` that checks for each different special form.

Instead, well use a dispatch dictionary, which we've called `SPECIAL_FORMS`. If you find `SPECIAL_FORMS`, you can see that the keys are the different commands and the values are the respective `do_*_form` function. Therefore, what would have been 20 lines of code was condensed to 2 lines of code. When the interpreter does `SPECIAL_FROMS[first]`, we get the `do_*_form` function out and then call it with the expression.

```
def scheme_eval(expr):
    ...
    first, rest = scheme_car(expr), scheme_cdr(expr)
    if (scheme_symbolp(first) and first in SPECIAL_FORMS):
        if proper_tail_recursion:
            "*** YOUR CODE HERE ***"
        else:
            expr, env = SPECIAL_FORMS[first](rest, env)
            expr, env = scheme_eval(expr, env), None
    ...
```

One important thing to note is that after you call `do_*_form`, you `scheme_eval` the result again. Why is that? Also what values can the `expr` and `env` (from `SPECIAL_FROMS[first]`) be? *Hint: there is more than one pair of expression, environment that you can return. *