Interpreters, Programs as Data & Macros Meta

Computer Science Mentors 61A

April 14 – April 18, 2025

Recommended Timeline

- Interpreters Intro (10-15 min): Consider walking through these problems as you give a mini-lecture/review to leave more time for later problems.
- Eval Apply (10 min): You may want to do one in its entirety before leaving students to try the rest (and explain the rules in a step-by-step manner), since students often find these types of problems unintuitive or challenging.
- Macros Intro / mini-lecture (5 min)
- `macros-quasi` (5 min): A WWSD-type problem.
- Macro Quasi [10 Mins]
- Meta Apply [5 Mins]
- NAND [5 Mins]
- Apply Twice [5 Mins]
- Censor [10 Mins]

Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback

1 Interpreters

An **interpreter** is a computer program that understands, processes, and executes other programs. The Scheme interpreter we will cover in CS 61A is built around the **Read-Eval-Print Loop**, which consists of the following steps:

1. **Read** the raw input and parse it into a data structure we can easily handle.
2. **Evaluate** the parsed expression.
3. **Print** the result to output.

One of the challenges of designing interpreters is to represent the input in a way that the interpreter's language can understand. For example, since our Scheme interpreter is written in Python, we need to parse Scheme tokens into a usable Python representation. Conveniently, every Scheme call expression and special form is represented in Scheme as a linked list. Therefore, we will represent Scheme lists with the `Pair` class, which is a type of linked list.

Just like `Link`, a `Pair` instance has two attributes, `first` and `second`, which contain the first element and rest of the linked list, respectively. Instead of using `Link.empty` to represent an empty list, `Pair` uses `nil`.

For example, during the read step, the Scheme expression `(+ 1 2)` would be **tokenized** into `'(', '+', '1', '2', ')'` and then organized into a `Pair` instance as `(Pair('+', Pair(1, Pair(2, nil))))`.

Once we have parsed our input, we evaluate the expression by calling `scheme_eval` on it. If it's a procedure call, we recursively call `scheme_eval` on the operator and the operands. Then we return the result of calling `scheme_apply` on the evaluated operator and operands, which computes the procedure call. If it's a special form, the relevant evaluation rules are followed in a similar matter.

For example, when we provide `(+ 1 (+ 2 3))` as input to the interpreter, the following happen:

- `(+ 1 (+ 2 3))` is parsed to `Pair('+', Pair(1, Pair(Pair('+', Pair(2, Pair(3, nil))), nil)))`
- The interpreter recognizes this is a procedure call.
- `scheme_eval` is called on the operator, `'+'`, and returns the addition procedure.
- `scheme_eval` is called on the operand `1` and returns `1`.
- `scheme_eval` is called on the operand `Pair('+', Pair(2, Pair(3, nil)))`.
  - The interpreter recognizes this is a procedure call.

  - `scheme_eval` is called on the operator, `'+'`, and returns the addition procedure.
  - `scheme_eval` is called on the operand `2` and returns `2`.
  - `scheme_eval` is called on the operand `3` and returns `3`.
  - `scheme_apply` is called on the evaluated procedure and parameters (`Pair(2, Pair(3, nil))`) and returns `5`.
- `scheme_apply` is called on the evaluated procedure and parameters (`Pair(1, Pair(5, nil))`) and returns `6`.
- `6` is printed to output.

Explaining interpreters can be quite tricky. You will probably need to field a lot of student questions in order to ensure that they are understanding things.

Everything in Scheme is lists!!!! I think this is something tricky that a lot of students don't really understand. Understanding this once and for all is what finally made Scheme make sense to me, but your mileage may vary.

Here are a few questions that you probably want to be answered by your mini-lecture:

1. The following questions refer to the Scheme interpreter. Assume we're using the implementation seen in lecture and in the Scheme project.
   (a) What's the purpose of the read stage in a Read-Eval-Print Loop? For our Scheme interpreter, what does it take in, and what does it return?

   The read stage returns a representation of the code that is easier to process later in the interpreter by putting it in a new data structure. In our interpreter, it takes in a string of code, and outputs a Pair representing an expression (which is really just the same as a Scheme list).

   (b) What are the two components of the read stage? What do they do?

   The read stage consists of

   1. The lexer, which breaks the input string and breaks it up into tokens (individual characters or symbols)
   2. The parser, which takes that string of tokens and puts it into the data structure that the read stage outputs (in our case, a Pair).

   (c) Write out the constructor for the `Pair` object that the read stage creates from the input string `(define (foo x) (+ x 1))`

   Pair("define", Pair(Pair("foo", Pair("x", nil)), Pair(Pair("+", Pair("x", Pair(1, nil))), nil)))

(d) For the previous example, imagine we saved that Pair object to the variable `p`. How could we check that the expression is a `define` special form? How would we access the name of the function and the body of the function?

We could check to see that it's a define special form by checking if `p.first == "define"`.

We could get its name by accessing `p.second.first.first` and get the body of the function with `p.second.second.first`.

Teaching Tips

- A great way to go about these short answer type questions is to have a mini lecture prepared and then go through the answer of each question in your lecture.
- Often the words read, eval, and print may not make the most intuitive sense to students right away. Encourage them to think about them in different angles and make analogies to listening to someone talk or following an instruction: you always process what you receive first, then you actually do the thing, and then you show that you understood or were able to produce results.
- If you can think of a clever way to remember lexer and parser that would be really helpful to students!
- Remind students that Pairs are nothing more than linked lists to lessen the possible apprehension at hand-creating the Pairs. This will also save a lot of headaches with .seconds and .firsts in the project. It may even be helpful to draw out an environment diagram of the Pair structure as a linked list.

2. Circle or write the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(if 1 (+ 2 3) (/ 1 0))

 scheme_eval   1  3  4  6
 scheme_apply  1  2  3  4
```

6 `scheme_eval`, 1 `scheme_apply`. Evals: (1) on the entire expression, (2) on `1` (**`if`** is not evaluated), (3) on (+ 2 3), (4-6) on +, 2, 3. Apply: (1) with applying + on (+ 2 3).

```
(or #f (and (+ 1 2) 'apple) (- 5 2))

 scheme_eval   6  8  9  10
 scheme_apply  1  2  3   4
```

8 `scheme_eval`, 1 `scheme_apply`.

```
(define (square x) (* x x))

(+ (square 3) (- 3 2))

 scheme_eval   2  5  14  24
 scheme_apply  1  2   3   4
```

14 `scheme_eval`, 4 `scheme_apply`.

```
(define (add x y) (+ x y))

(add (- 5 3) (or 0 2))
```

13 `scheme_eval`, 3 `scheme_apply`.

Teaching Tips

- This has historically been a tricky concept for students. scheme_apply may come off as easier to understand so relate it to just applying operators to operands for students.
- Remind students of what types of expressions will get scheme_eval'ed: parenthetical expressions, operators, function names, and special key words.
- Remind students that in the case of special forms, there is scheme_apply – each special form has their own way of handling their arguments.
- Be very conscious about not accidentally evaluating the expressions yourself when you are counting; that's the interpreter's job!
- It might help to count the expressions by scheme_apply groups, i.e. count all of scheme_eval for one scheme_applyable group and then move onto the next.
- Consider referencing Josh's helpful walkthrough video.

2 Macros

Up to now, all of our programs have taken data—integers, strings, numbers, lists, and more—and manipulated this data to accomplish a wide variety of tasks. What if computer code itself could serve as the data used and produced by programs? What if we could treat **programs as data**?

We've already seen something similar when we built higher order functions. Functions can be thought of as "bundles of code," and HOFs allowed us to treat this code as input and output to other functions. For example, the `twice` function below takes in a zero-argument function `f` and calls `f` twice.

```
def twice(f):
    f()
    f()

>>> twice(lambda:
   print(5))
5
5
```

```
(define (twice f)
    (f)
    (f))

scm> (twice (lambda () (print 5)))
5
5
```

Beyond HOFs, Scheme gives us many more tools for the treatment of programs as data.

It's worth asking your students why this function is different from something like this:

```
(define (twice arg)
    arg
    arg)

scm> (twice (lambda () (print 5)))
5
```

The answer, of course, is that the expression written in the operand slot of `twice` is only evaluated once, whereas using functions to "wrap" this expression has allowed us to delay evaluation until we want it.

2.1 Quotation

Recall that Scheme is a list processing language. The call expression `(+ 1 2)` is a list literally consisting of the symbol `+` and the numbers `1` and `2`. When this list is provided the interpreter, the appropriate evaluation rules are followed and the expression evaluates to `3`. Similarly, when we provide the list `(1 2 3)` to the Scheme interpreter, it attempts to evaluate the list as a call expression and encounters an error. In Scheme, there is no distinction between lists and call expressions/special forms.

Since Scheme programs are essentially stored as a long list, if we are to use programs as input to other programs, we need a way to prevent them from being evaluated while we manipulate them. The `quote` special form, also denoted by an apostrophe ´, which simply returns its unevaluated operand:

```
scm> '(+ 1 2)
(+ 1 2)
scm> (list '(if #t (\ 1 0) 3)(+ 2 3))
((if #t (\ 1 0) 3) 5)
```

Quotation is a "protective shell" that prevents the immediately following expression from being evaluated as it passes through the interpreter.

On the other hand, **`eval`** is a procedure that simply evaluates its argument. Note that since **`eval`** is a procedure, its argument is evaluated first before applying **`eval`**. Whereas quotation prevents evaluation, **`eval`** evaluates things another time.

```
scm> (eval '(+ 1 2))
3
scm> (eval (list 1 2 3))
Error: int is not callable
```

Quotation allows us to begin taking code as input. For example, the following version of `twice` takes in an expression and evaluates it twice:

```
(define (twice expr)
    (eval expr)
    (eval expr))

scm> (twice '(print 5))
5
5
```

2.2 Quasiquotation

The quasiquote special form, denoted with a backtick `` ` ``, has the same effect as `'`, except that any subexpressions can be "unquoted" by preceding them with a comma `,`. Any unquoted subexpression is evaluated as normal, whereas everything else is left unevaluated.

```
(define (cool-string tens-digit ones-digit letter)
    (I love ,tens_digit ,ones_digit ,letter))

scm> (cool-string 6 1 'a)
(i love 6 1 a)
```

This is very similar to f-string behavior in Python:

```
def cool_string(tens_digit, ones_digit, letter):
    return f"I love {tens_digit}{ones_digit}{letter}"

>>> cool_string(6, 1, "a")
'I love 61a'
```

The analogy is summed up by the following:

- Quotation `'...` in Scheme is like strings `"..."` in Python
- Quasiquotation `` `... `` in Scheme is like f-strings in Python `f"..."`
- Unquotation `,...` in Schemes is like replacement fields `{...}` in Python

2.3 Macros

A call expression in Scheme is evaluated by evaluating the operator, then evaluating the operands, before finally applying the operator to the operands. Because the parameters of a Scheme procedure are evaluated before the body of the procedure is evaluated, we say that procedures operate on values.

In Scheme, a **macro** is similar to a procedure, but it operates on expressions rather than value. Thus the input to macros is code that we can manipulate as data. Macro evaluation involves three steps:

1. Evaluate the operator to a macro procedure.
2. Apply the macro procedure to the *unevaluated* operands
3. Evaluate the expression produced by the macro procedure in the same frame it was called in and return the result.

This may sound overwhelming at first, but just remember that the key difference between macro procedures and regular procedures are that 1) in macros, operands are not evaluated and 2) after the body of the macro produces an expression, the expression is automatically evaluated one more time.

New macros are defined using the special form **`define-macro`**. Below is an example of a macro `twice` that evaluates a given expression twice.

```
(define-macro (twice expr)
    (list 'begin expr expr))

scm> (twice (print 'hello))
hello
hello
```

When `twice` is called, the unevaluated expression `'(print 'hello)` is bound to the symbol `expr`. The body of the macro is executed as normal, producing the expression `(begin (print 'hello)(print 'hello))`. Finally, this expression is evaluated in the global frame, and the macro call prints `hello` twice.

Note that even though we pass in `(print 'hello)` as an operand, we don't evaluate the expression and print right away. Because the body is evaluated without evaluating the operands at first, macros allow us to implement new special forms, control the order of evaluation, and more.

1. What will Scheme output?

```
scm> (define x 6)

x
scm> (define y 1)

y
scm> '(x y a)

(x y a)
scm> `(,x ,y a)

(6 1 a)
scm> `(,x y a)

(6 y a)
scm> `(,(if (< 1 2) '+ '-) 1 2)

(+ 1 2)
scm> (eval `(,(if (< 1 2) '+ '-) 1 2))

3
scm> (define (add-expr a1 a2)
               (list '+ a1 a2))

add-expr
scm> (add-expr 3 4)

(+ 3 4)
```

```
scm> (eval (add-expr 3 4))

7

scm> (define-macro (add-macro a1 a2)
            (list '+ a1 a2))

add-macro

scm> (add-macro 3 4)

7
```

2. The built-in `apply` procedure in Scheme applies a procedure to a given list of arguments. For example, `(apply f '(1 2 3))` is equivalent to `(f 1 2 3)`. Write a macro procedure `meta-apply`, which is similar to `apply`, except that it works not only for procedures, but also for macros and special forms. That is, `(meta-apply operator (operand1 ... operandN))` should be equivalent to `(operator operand1 ... operandN)` for any operator and operands. See doctests for examples.

```
; Doctests
scm> (meta-apply + (1 2))
3
scm> (meta-apply or (#t (/ 1 0) #f))
#t
(define-macro (meta-apply operator operands)


)


(define-macro (meta-apply operator operands)
    (cons operator operands))
```

This is supposed to be a relatively gentle introduction to the process of writing macros. Nothing much to see here. Though there may be some students who want to just write `(operator operands)`. The issue with this, of course, is that it is treated as a call expression in the body of the macro, which causes an error. An alternate solution with quasiquote is also possible:

```
(define-macro (meta-apply operator operands)
    `(,operator ,operands))
```

3. NAND (not and) is a logical operation that returns false if all of its operands are true, and true otherwise. That is, it returns the opposite of AND. Implement the `nand` macro procedure below, which takes in a list of expressions and returns the NAND of their values. Similar to **`and`**, `nand` should short circuit and return true as soon as it encounters a false operand, evaluating from left to right.

   Hint: You may use `meta-apply` in your implementation.

```
;Doctests
scm> (nand (#t #t #t #t #t #t))
#f
scm> (nand (#t #f #t))
#t
scm> (nand (#f (/ 1 0)))
#t
(define-macro (nand operands))


)
```

```
(define-macro (nand operands)
    `(not (meta-apply and ,operands)))
```

   This problem must be completed after `meta-apply` because the solution involves `meta-apply`.

   If students are having trouble with this problem, first try to make sure that they have a good understanding of how `nand` works. You can walk though the problem, or draw them a table to show them the values of `nand` with different arrangements of true and false.

   A common issue with macro problems is that our Scheme interpreter does not allow us to define procedures and macros that take arbitrary numbers of elements. Therefore, `nand` must take a list of operands rather than taking the operands directly. This is a difference that you should note for your students (though you don't have to explain to them why.)

   A good way to think of this problem is to ask your students: "How could you logically NAND something without defining a function? What sequence of operations would allow you to do that?" The answer, of course, is that you would take the **`and`** of the expressions and the apply **`not`** to the result, i.e., `(not (and a b))`. Therefore, the body of your macro procedure should probably evaluate to an expression that looks something like that.

The next hiccup is how you apply **and** to a list of operands; hopefully the hint will guide them to the conclusion that they should employ the previously defined `meta-apply`, but if not you can ask them a leading question to that effect.

4. Implement `apply-twice`, which is a macro that takes in a call expression with a single argument. It should return the result of applying the operator to the operand twice.

```
;Doctests
scm> (define add-one (lambda (x) (+ x 1)))
add-one
scm> (apply-twice (add-one 1))
3
scm> (apply-twice (print 'hi))
hi
undefined

(define-macro (apply-twice call-expr)
                  ________________________
)


(define-macro (apply-twice call-expr)
  (list (car call-expr) call-expr)
)
```

Teaching Tips

- The first question to ask your students would be, "What are the inputs to our function?" In this case, we are accepting a scheme list called call-expr containing an operator and an operand.
- The second question to ask your students would be, "How should our function behave?" We want to apply the operator onto the result of applying the operator to the operand. So applying our function twice, so for an arbitrary function `f`, and input `x`, `f(f(x))`
- So how do we apply both of these. We create the list! The expression should have two elements, the first one being the operator of `call-expr`, and the second being another list, with the operator and operand of `call-expr`
- if your students ask why this has to be a macro procedure, consider running through the function normally to see that you would have to initially evaluate the operands of the function, which wouldn't be particularly useful. For example if we had the call `(apply-twice (add-one 3))` with a function `add-one` that adds one to our input, we would simply pass in 4 into our function `apply-twice` which isn't that helpful

5. Write a macro procedure `censor`, which takes in an expression `expr` and a symbol `phrase`. If `expr` does not contain any instance of `phrase`, then `censor` simply evaluates `expr`. However, if `expr` does contain an instance of the censored phrase, the symbol `censored` is returned and the expression is not evaluated.

```
;Doctests
scm> (censor ((lambda (stanford tree) (+ stanford tree)) 4 5)
   stanford)
censored
scm> (censor ((lambda (stanford tree) (+ stanford tree)) 4 5)
   tree)
censored
scm> (censor ((lambda (stanford tree) (+ stanford tree)) 4 5)
   ree)
9

(define-macro (censor expr phrase)
    (define (contains-phrase expr)




    )
    (if ____________________________________________

        ____________________________________________

        ____________________________________________))
```

```
(define-macro (censor expr phrase)
    (define (contains-phrase expr)
        (cond
            ((equal? expr phrase) #t)
            ((or (not (list? expr)) (null? expr)) #f)
            (else (or (contains-phrase (car expr))
               (contains-phrase (cdr expr))))))
    (if (contains-phrase expr)
        ''censored
        expr))
```

There are many, many ways to complete the `contains-phrase` helper procedure, which is why no skeleton code was offered for that portion. Once the helper procedure is defined, the problem is relatively straightforward, except for the double quote on the censored. We expect that very few students will recognize the need for a double quote there, and that's ok. When they have questions about it, you should note that two quotes are needed because the return expression is evaluated once within the body of the macro and then again as it is returned.

The `contains-phrase` helper procedure is a relatively straightforward recursive procedure. Note that it is a procedure, not a macro. If students are confused about how to approach the design of the procedure, you can help lead them toward the fact that `expr` is just a bunch of nested lists, and they just need to determine if the `phrase` is contained somewhere in those nested lists.