Scheme Review and Lists Solutions

Computer Science Mentors 61A

November 18 – November 23, 2024

1 Interpreters

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`.

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`.

2 Scheme Macros

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))
```

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)))
```

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)
)
```

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))
```