CS 61A Spring 2022

Programs as Data

Discussion 12: April 13, 2022 Solutions

##Walkthrough Videos

Feel free to try these problems on the worksheet in discussion or on your own, and then come back to reference these walkthrough videos as you study.

To see these videos, you should be logged into your berkeley.edu email.

Scheme Programs as Data

All Scheme programs are made up of expressions. There are two types of expressions: primitive expressions and combinations.

- Primitive expression examples: `#f`, `1.7`, `+`
- Combinations examples: `(fact 10)`, `(/ 8 3)`, `(not #f)`

Scheme's built-in list data structure can be used to represent combinations.

- Example: `(list 'fact 10)` results in the combination `(fact 10)`.

Quasiquotation

The normal quote `'` and the quasiquote `` ` `` are both valid ways to quote an expression. However, the quasiquoted expression can be *unquoted* with the "unquote" `,` (represented by a comma). When a term in a quasiquoted expression is *unquoted*, the unquoted term is *evaluated*.

```
scm> (define a 5)
a
scm> (define b 3)
b
scm> `(* a b)
(* a b)
scm> `(* a ,b)
(* a 3)
scm> '(* a ,b)
(* a (unquote b))
```

Q1: WWSD? Quasiquotation

```
scm> '(1 x 3)
```

(1 x 3)

```
scm> (define x 2)
```

x

```
scm> `(1 x 3)
```

(1 x 3)

```
scm> `(1 ,x 3)
```

(1 2 3)

```
scm> '(1 ,x 3)
```

(1 (unquote x) 3)

```
scm> `(,1 x 3)
```

(1 x 3)

```
scm> `,(+ 1 x 3)
```

6

```
scm> `(1 (,x) 3)
```

(1 (2) 3)

```
scm> `(1 ,(+ x 2) 3)
```

(1 4 3)

```
scm> (define y 3)
```

y

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

(x 6 y)

```
scm> `(1 ,(cons x (list y 4)) 5)
```

(1 (2 3 4) 5)

Eval Procedure

The `eval` procedure forces evaluation of a given expression in the current environment. Since a quote supresses evaluation, calling `eval` on a quoted expression `(quote expr)` will evaluate the expression `expr`.

```
scm> (define a '(1 2 3))
a
scm> (quote a) ; equivalently, 'a
a
scm> (eval 'a)
(1 2 3)
```

Apply Procedure

When evaluating an expression, once the `operator` and `operands` have been fully evaluated, the operator is `apply`'d using the operands as arguments. This can also be done outside of the implicit context of evaluations using the `apply` procedure. The `apply` procedure applies a given `operator` to a list of `operands`.

```
scm> (apply + '(2 3))
5
scm> (apply (lambda (x) (* 2 x)) (list 1))
2
```

Q2: WWSD? Eval and Apply

```
scm> (define add-numbers '(+ 1 2))
```

add-numbers

```
scm> add-numbers
```

(+ 1 2)

```
scm> (eval add-numbers)
```

3

```
scm> (apply + '(1 2)) ; Is this similar to the previous eval call?
```

3

```
scm> (define expr '(lambda (a b) (+ a b)))
```

expr

```
scm> expr
```

(lambda (a b) (+ a b))

```
scm> (define adder-func (eval expr))
```

adder-func

```
scm> (apply adder-func '(1 2))
```

3

```
scm> (define make-list (cons 'list '(1 2 3)))
```

make-list

```
scm> make-list
```

(list 1 2 3)

```
scm> (eval make-list)
```

(1 2 3)

```
scm> (apply list '(1 2 3)) ; Is this similar to the previous eval
    call?
```

(1 2 3)

Q3: Geometric Sequence

Implement the procedure `geom`, which takes in a nonnegative integer `n` and a factor `f` that is an integer greater than 0. The procedure should create a program as a list that, when passed into the `eval` procedure, evaluates to the `n`th number of the geometric sequence that starts at 1 and has a factor of `f`. The sequence is zero-indexed.

For example, the geometric sequence starting at 2 is 1, 2, 4, 8, and so on. The expression `(geom 5 2)` returns a program as a list. When `eval` is called on that returned list, it should evaluate to the 5th number of the geometric sequence that has a factor of 2 (and starts at 1), which is 32.

```
(define (geom n f)
    (if (= n 0) 1
        (list '* (geom (- n 1) f) f)))

(define expr (geom 1 5))
(expect expr (* 1 5))
(expect (eval expr) 5)

(define expr2 (geom 2 5))
(expect expr2 (* (* 1 5) 5))
(expect (eval expr2) 25)
```

Q4: Make Or

Implement `make-or`, which returns, as a list, a program that takes in two expressions and `or`'s them together (applying short-circuiting rules). However, do this without using the `or` special form. You may also assume the name `v1` doesn't appear anywhere outside this function. For a quick reminder on the short-circuiting rules for `or` take a look at slide 18 of Lecture 3 on Control.

The behavior of the `or` procedure is specified by the following doctests:

```
scm> (define or-program (make-or '(print 'bork) '(/ 1 0)))
or-program
scm> (eval or-program)
bork
scm> (eval (make-or '(= 1 0) '(+ 1 2)))
3
```

```
(define (make-or expr1 expr2)
    `(let ((v1 ,expr1))
        (if v1 v1 ,expr2))
)
```

Q5: Make "Make Or"

The above code generates a program that evaluates an `or` expression without using any `or` statements. However, we can take it even one step further: let's create a program which generates `make-or`, the program you created which generates an `or` expression.

Implement `make-make-or`, a program which generates a program which, when `eval`'d, can be `apply`'d to make an `or` expression with differing varibles. You may find the code you wrote above to be useful.

> Hint: recall that you want to construct a list that resembles the program. Do you know what this list would look like?

```
(define (make-make-or)
  '(define (make-or expr1 expr2) `(let ((v1 ,expr1)) (if v1 v1 ,
    expr2)))
)
```

Now, given this function, determine the outputs from the following expressions:

```
scm> (make-make-or)
```

(define (make-or expr1 expr2) (quasiquote (let ((v1 (unquote expr1))) (if v1 v1 (unquote expr2)))))

```
scm> (eval (make-make-or))
```

make-or

```
scm> (eval (eval (make-make-or)))
```

(lambda (expr1 expr2) (quasiquote (let ((v1 (unquote expr1))) (if v1 v1 (unquote expr2)))))

```
scm> (apply (eval (eval (make-make-or))) '(#t (/ 1 0)))
```

(let ((v1 #t)) (if v1 v1 (/ 1 0)))

```
scm> (eval (apply (eval (eval (make-make-or))) '(#t (/ 1 0))))
```

#t