Lab 11: Programs as Data, Macros

lab11.zip (lab11.zip)

Due by 11:59pm on Wednesday, April 17.

Starter Files

Download lab11.zip (lab11.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Required Questions

Getting Started Videos

Quasiquotation

Consult the drop-down if you need a refresher on quasiquotation. It's okay to skip directly to the questions and refer back here should you get stuck.

Quasiquotation

Q1: WWSD: Quasiquote

Use Ok to test your knowledge with the following "What Would Scheme Display?" questions:

```
python3 ok -q wwsd-quasiquote -u
```

```
scm> '(1 x 3)

scm> (define x 2)

scm> `(1 x 3)

scm> `(1 ,x 3)

scm> `(1 x ,3)

scm> `(1 (,x) 3)

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

scm> (define y 3)

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

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

Programs as Data

Consult the drop-down if you need a refresher on Programs as Data. It's okay to skip directly to the questions and refer back here should you get stuck.

Programs as Data

Q2: If Program

In Scheme, the `if` special form allows us to evaluate one of two expressions based on a predicate. Write a program `if-program` that takes in the following parameters:

1. `predicate` : a quoted expression which will evaluate to the condition in our `if` -expression
2. `if-true` : a quoted expression which will evaluate to the value we want to return if `predicate` evaluates to true (`#t`)
3. `if-false` : a quoted expression which will evaluate to the value we want to return if `predicate` evaluates to false (`#f`)

The program returns a Scheme list that represents an `if` expression in the form: `(if <predicate> <if-true> <if-false>)` . Evaluating this expression returns the result of evaluating this `if` expression.

Here are some doctests to show this:

```
scm> (define x 1)
scm> (if-program '(= 0 0) '(+ x 1) 'x)
(if (= 0 0) (+ x 1) x)
scm> (eval (if-program '(= 0 0) '(+ x 1) 'x))
2
scm> (if-program '(= 1 0) '(print 3) '(print 5))
(if (= 1 0) (print 3) (print 5))
scm> (eval (if-program '(= 1 0) '(print 3) '(print 5)))
5
```

```
(define (if-program condition if-true if-false)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q if-program
```

Q3: Exponential Powers

Implement a procedure `(pow-expr base exp)` that returns an expression that, when evaluated, raises the number `base` to the power of the nonnegative integer `exp` . The body of `pow-expr` should not perform any multiplication (or exponentiation). Instead, it should just construct an expression containing only the symbols `square` and `*` as well as the number `base` and parentheses. The length of this expression should grow logarithmically with respect to `exp` , rather than linearly.

Examples:

```
scm> (pow-expr 2 0)
1
scm> (pow-expr 2 1)
(* 2 1)
scm> (pow-expr 2 5)
(* 2 (square (square (* 2 1))))
scm> (pow-expr 2 15)
(* 2 (square (* 2 (square (* 2 (square (* 2 1)))))))
scm> (pow-expr 2 16)
(square (square (square (square (* 2 1)))))
scm> (eval (pow-expr 2 16))
65536
```

> *Hint:*
>
> 1. $x^{2y} = (x^y)^2$
> 2. $x^{2y+1} = x(x^y)^2$
>
> For example, $2^{16} = (2^8)^2$ and $2^{17} = 2 * (2^8)^2$.
>
> You may use the built-in predicates `even?` and `odd?`. Also, the `square` procedure is defined for you.

Here's the solution to a similar homework problem (https://cs61a.org/hw/sol-hw07/#q1-pow).

```
(define (square n) (* n n))

(define (pow-expr base exp)
    'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q pow
```

Macros

A macro is a code transformation that is created using `define-macro` and applied using a call expression. A macro call is evaluated by:

1. Binding the formal paramters of the macro to the **unevaluated** operand expressions of the macro call.
2. Evaluating the body of the macro, which returns an expression.

3. Evaluating the expression returned by the macro in the frame of the original macro call.

```
scm> (define-macro (twice expr) (list 'begin expr expr))
twice
scm> (twice (+ 2 2))  ; evaluates (begin (+ 2 2) (+ 2 2))
4
scm> (twice (print (+ 2 2)))  ; evaluates (begin (print (+ 2 2)) (print (+ 2 2)))
4
4
```

Q4: Repeat

Define `repeat`, a macro that is called on a number `n` and an expression `expr`. Calling it evaluates `expr` in a local frame `n` times, and its value is the final result. You will find the helper function `repeated-call` useful, which takes a number `n` and a zero-argument procedure `f` and calls `f` `n` times.

For example, `(repeat (+ 2 3) (print 1))` is equivalent to:

```
(repeated-call (+ 2 3) (lambda () (print 1)))
```

and should evaluate `(print 1)` repeatedly 5 times.

The following expression should print `four` four times:

```
(repeat 2 (repeat 2 (print 'four)))
```

```
(define-macro (repeat n expr)
  `(repeated-call ,n ___))

; Call zero-argument procedure f n times and return the final result.
(define (repeated-call n f)
  (if (= n 1) ___ (begin ___ ___)))
```

Use Ok to test your code:

```
python3 ok -q repeat-lambda
```

Hint: repeat

Hint: repeated-call