Homework 9: Programs as Data, Macros

hw09.zip (hw09.zip)

Due by 11:59pm on Tuesday, November 28

Instructions

Download hw09.zip (hw09.zip). Inside the archive, you will find a file called hw09.scm (hw09.scm), along with a copy of the `ok` autograder.

Submission: When you are done, submit the assignment by uploading all code files you've edited to Gradescope. You may submit more than once before the deadline; only the final submission will be scored. Check that you have successfully submitted your code on Gradescope. See Lab 0 (/lab/lab00#submitting-the-assignment) for more instructions on submitting assignments.

Using Ok: If you have any questions about using Ok, please refer to this guide. (/articles/using-ok)

Readings: You might find the following references useful:

- Scheme Specification (/articles/scheme-spec/)
- Scheme Built-in Procedure Reference (/articles/scheme-builtins/)

Grading: Homework is graded based on correctness. Each incorrect problem will decrease the total score by one point. There is a homework recovery policy as stated in the syllabus. **This homework is out of 2 points.**

Macros are a method of programming that allow programmers to treat expressions as data and create procedures of a language using the language itself. Macros open the door to many clever tricks of creating "shortcuts", and with Scheme, allow us to build our own special forms other than the ones that are built in.

Required Questions

Getting Started Videos

Programs as Data: Chef Curry

Recall that currying transforms a multiple argument function into a series of higher-order, one argument functions. In the next set of questions, you will be creating functions that can automatically curry a function of any length using the notion that programs are data!

Q1: Cooking Curry

Implement the function `curry-cook`, which takes in a Scheme list `formals` and a quoted expression `body`. `curry-cook` should generate a program as a list which is a curried version of a lambda function. The outputted program should be a curried version of a lambda function with formal arguments equal to `formals`, and a function body equal to `body`. You may assume that all functions passed in will have more than 0 `formals`; otherwise, it would not be curry-able!

For example, if you wanted to curry the function `(lambda (x y) (+ x y))`, you would set `formals` equal to `'(x y)`, the `body` equal to `'(+ x y)`, and make a call to `curry-cook`: `(curry-cook '(x y) '(+ x y))`.

```
scm> (curry-cook '(a) 'a)
(lambda (a) a)
scm> (curry-cook '(x y) '(+ x y))
(lambda (x) (lambda (y) (+ x y)))
```

```
(define (curry-cook formals body)
    'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q curry-cook
```

Q2: Consuming Curry

Implement the function `curry-consume`, which takes in a curried lambda function `curry` and applies the function to a list of arguments `args`. You may make the following assumptions:

1. If `curry` is an `n`-curried function, then there will be at most `n` arguments in `args`.
2. **If there are 0 arguments** (`args` is an empty list), then you may assume that `curry` has been fully applied with relevant arguments; in this case, `curry` now contains a value representing the output of the lambda function. Return it.

Note that there can be fewer `args` than `formals` for the corresponding lambda function `curry`! In the case that there are fewer arguments, `curry-consume` should return a curried lambda function, which is the result of partially applying `curry` up to the number of `args` provdied. See the doctests below for a few examples.

```
scm> (define three-curry (lambda (x) (lambda (y) (lambda (z) (+ x (* y z))))))
three-curry
scm> (define eat-two (curry-consume three-curry '(1 2))) ; pass in only two arguments, ret
eat-two
scm> eat-two
(lambda (z) (+ x (* y z)))
scm> (eat-two 3) ; pass in the last argument; 1 + (2 * 3)
7
scm> (curry-consume three-curry '(1 2 3)) ; all three arguments at once
7
```

```
(define (curry-consume curry args)
    'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q curry-consume
```

Macros

Q3: Switch to Cond

`switch` is a macro that takes in an expression `expr` and a list of pairs, `cases`, where the first element of each pair is some value and the second element is a single expression. `switch` evaluates the expression contained in the list of `cases` that corresponds to the value that `expr` evaluates to.

```
scm> (switch (+ 1 1) ((1 (print 'a))
                      (2 (print 'b)) ; (print 'b) is evaluated because (+ 1 1) evaluates t
                      (3 (print 'c))))
b
```

`switch` uses another procedure called `switch-to-cond` in its implementation:

```
scm> (define-macro (switch expr cases)
                   (switch-to-cond (list 'switch expr cases))
     )
```

Your task is to define `switch-to-cond`, which is a procedure (not a macro) that takes a quoted `switch` expression and converts it into a `cond` expression with the same behavior. An example is shown below.

```
scm> (switch-to-cond `(switch (+ 1 1) ((1 2) (2 4) (3 6))))
(cond ((equal? (+ 1 1) 1) 2) ((equal? (+ 1 1) 2) 4) ((equal? (+ 1 1) 3) 6))
```

```
(define-macro (switch expr cases) (switch-to-cond (list 'switch expr cases)))

(define (switch-to-cond switch-expr)
  (cons _________
    (map
          (lambda (case) (cons _________________ (cdr case)))
          (car (cdr (cdr switch-expr))))))
```

Use Ok to test your code:

```
python3 ok -q switch-to-cond
```

Q4: Factor Switch

Note: This question assumes you finished implementing `switch-to-cond` in the previous problem.

Define the procedure `switch-factors`, which uses the `switch` macro to determine whether a number is one, prime, or composite.

- A prime number `n` is a number that is not divisible by any numbers other than 1 and `n` itself.
- A composite number `n` is a number that is divisible by at least one number other than 1 and `n`.

```
scm> (switch-factors 1)
one
scm> (switch-factor 17)
prime
scm> (switch-factor 9)
composite
```

You may use the `min`, `count`, and `is-factor` procedures, which have already been defined for you.

> **Hint**: `switch` doesn't have an `else` case. In other words, `switch` only returns an expression when `expr` equals the value correponding to that expression. Knowing this, how can the `min` procedure be useful?

```
(define (min x y) (if (< x y) x y))

(define (count f n i) (if (= i 0) 0 (+ (if (f n i) 1 0) (count f n (- i 1)))))

(define (is-factor dividend divisor) (if (equal? (modulo dividend divisor) 0) #t #f))

(define (switch-factors n)
  (switch _________ ___________________)
)
```

Use Ok to test your code:

```
python3 ok -q switch-factors
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit

Make sure to submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** For a refresher on how to do this, refer to Lab 00 (https://cs61a.org/lab/lab00/#submit-with-gradescope).

Exam Practice

Homework assignments will also contain prior exam questions for you to try. These questions have no submission component; feel free to attempt them if you'd like some practice!

Macros

1. Fall 2019 Final Q9: Macro Lens (https://cs61a.org/exam/fa19/final/61a-fa19-final.pdf#page=10)
2. Summer 2019 Final Q10c: Slice (https://cs61a.org/exam/su19/final/61a-su19-final.pdf#page=10)
3. Spring 2019 Final Q8: Macros (https://cs61a.org/exam/sp19/final/61a-sp19-final.pdf#page=8)