Writing Macros

Normal Scheme Procedures vs. Macros

```
(define (normal x)
  (list '+ x x))
```

```
(define-macro (macro x)
  (list '+ x x))
```

Above we have defined a normal Scheme procedure and a Scheme macro, both with the same formal parameters and body.

When we call `(normal (+ 1 2))`, first the argument `(+ 1 2)` is evaluated to 3. `x` is then bound to 3, so evaluating the body `(list '+ x x)` gives us the list `(+ 3 3)`.

When we call `(macro (+ 1 2))`, `x` is bound to the list `(+ 1 2)`, so evaluating the body `(list '+ x x)` gives us the list `(+(+12)(+12))`. We then evaluate this again to get 6.
Macros do not have their arguments evaluated, whereas normal Scheme procedures do. Macros have their return value re-evaluated a second time, whereas normal Scheme procedures do not.

Normal Scheme procedures take in values and return values. Macros can be seen as taking in code and returning code. Their arguments are not evaluated because the purpose of a macro is to work directly with the arguments’ code. Their return values are re-evaluated a second time because it is assumed that they return code, which should then be evaluated to an actual value.

**Macro-Writing as Code Rewriting**

When writing a macro, ask yourself this:

> What code do I wish the user had written instead of calling my macro?

If you can figure that out, and return that code from your macro, you are done.

Suppose we want to define a macro that will define a list-reduction procedure. Specifically, we want to provide a procedure name, an initial value, and a reduction operation, and have the macro define a matching list reduction procedure. An example will make this clearer.

```scheme
(define-macro (define-list-reducer name initial op)
  (define (product ___________)
    ____________
  )

(define-list-reducer product 1 *); defines product
>>(product '(1 3 5))
  15
>>(product nil)
  1
(define-list-reducer sum 0 +); defines sum
>>(sum '(1 3 5))
  9
>>(sum nil)
  0
```

Our goal is to define the macro `define-list-reducer`. So what code do we wish the user had written instead of `(define-list-reducer product 1 *)`? Well, to start, we wish the user had just defined themselves a procedure named `product` instead of making us implement a macro to do it.

```scheme
(define (product ___________)
  ____________
)
```

This is what we wish they had written... but how to fill it in?
(define (product lst)
    
)

Well, their product procedure should take in a single list as an argument.

(define (product lst)
    (if (null? lst)
        
        
    )
)

Their product procedure should make the empty list its base case.

(define (product lst)
    (if (null? lst)
        1
        
    )
)

If \textit{lst} is empty, then their result should be 1.

(define (product lst)
    (if (null? lst)
        1
        
    )
)

If \textit{lst} is not empty, then they should multiply the first number and the product of the rest of the numbers.

(define (product lst)
    (if (null? lst)
        1
        (*
            (car lst)
            (product (cdr lst))
        )
    )
)

This is what we wish they had written.
(define (list-reducer-macro name initial op)
 '(
 (define (name lst)
    (if (null? lst)
       initial
       (op
        (car lst)
        (name (cdr lst)))
    )
  )
)
)

For a first attempt at filling in the body of list-reducer-macro, we can try simply quoting the code we want to return. But what to replace name, initial, and op with?

(define (list-reducer-macro name initial op)
 '(
 (define (name lst)
    (if (null? lst)
       initial
       (op
        (car lst)
        (name (cdr lst)))
    )
  )
)

If we replace them with name, initial, and op, respectively, then due to the quote, the code we return will say literally name, initial, and op. But this is not what we
meant. We wanted to insert the values of name (eg. product), initial (eg. 1), and op (eg. *).

(define (list-reducer-macro name initial op)
  `(define (,name lst)
      (if (null? lst)
          ,initial
          ,(op
              (car lst)
              ,(name (cdr lst))
          )
      )
  )
)

The solution is to use a quasiquote instead of a quote and to replace <name>, <initial>, and <op> with ,name, ,initial, and ,op. The quasiquote (backtick) does the same thing as the quote (quotation mark) while additionally enabling the use of unquotes (commas) within. Unquotes cause the expression following them to be evaluated, so that ,name is replaced with product, ,initial is replaced with 1, and ,op is replaced with *.

(define-list-reducer
  product 1 *)

Now, when someone calls define-list-reducer, their call will effectively be replaced with the code returned by define-list-reducer. So, when they write (define-list-reducer product 1 *), they are effectively defining a recursive list-processing function named product that starts with the initial value 1 and *’s each element of the list. Above, the code on the left is replaced by the code on the right.

List Manipulation

Writing macros often involves complex list manipulation. Parts of the arguments (Scheme lists representing code) may need to be accessed. The return value (another Scheme list representing code) needs to be constructed. What follows is a review of some common list manipulation constructs in Scheme, along with their Python equivalents. These may be required for writing some macros.
### Practice Problems

**Case Special Form**

Write a macro that creates a case special form. The case special form has the following format.

```
(case
  <value>
  (
    (<value list 1> <result 1>)
    (<value list 2> <result 2>)
    (else <result 3>)
  )
)
```

First, `<value>` is evaluated. If `<value list 1>` contains `<value>`, then the result is `<result 1>`. Otherwise, if `<value list 2>` contains `<value>`, then the result is `<result 2>`. Otherwise, the result is `<result 3>`. An example usage follows.
(define x '(1 2 3))
(case
  (car (cdr (cdr x)))
    (
      ('(1 3 5) 'odd)
      ('(2 4 6) 'even)
      (else 'neither)
    )
)

The value of this case special form would be odd.

You may assume that a procedure called contains has been written for you. contains returns whether a given value is contained within a given list of values. It has the following signature: (contains value lst).

A hint appears on the last page of this packet, if desired.

(define-macro (case value clauses)
Hints

Case Special Form

We wish the caller had just written a cond expression instead of making us define a case macro. Return from case the code for a cond expression that would have the same effect as the case special form.

<value list 1> is accessible as (car (car clauses)).

<result 1> is accessible as (car (cdr (car clauses)).
Solutions

Case Special Form

(define-macro (case value clauses)
  `(cond
    (contains
      value
      ,(car (car clauses))
    ,(car (cdr (car clauses))))
    ,(car (cdr (car (cdr clauses))))
    (contains
      value
      ,(car (car (cdr clauses)))))
    ,(car (cdr (car (cdr (cdr clauses)))))))
  )))