Discussion 13: Macros

This is an online worksheet that you can work on during discussions and tutorials. Your work is not graded and you do not need to submit anything.

Discussion 13 Vitamin

To encourage everyone to watch or attend discussion orientation, we have created small discussion vitamins. Each vitamin you complete will give you an extra credit point towards the course. Please answer all of the questions in this form (https://links.cs61a.org/vitamin-for-disc13) after discussion orientation and before tutorial. If you have tutorial right after discussion, please complete within 3 hours after.

Macros

Previously, we've seen how we can create macro-like functions in Python using f-strings. Now let's talk about real macros, in Scheme. So far we've been able to define our own procedures in Scheme using the `define` special form. When we call these procedures, we have to follow the rules for evaluating call expressions, which involve evaluating all the operands.

In the scheme project, we saw that special form expressions do not follow the evaluation rules of call expressions. Instead, each special form has its own rules of evaluation, which may include not evaluating all the operands. Wouldn't it be cool if we could define our own special forms where we decide which operands are evaluated? Consider the following example where we attempt to write a function that evaluates a given expression twice:

```
scm> (define (twice f) (begin f f))
twice
scm> (twice (print 'woof))
woof
```

Since `twice` is a regular procedure, a call to `twice` will follow the same rules of evaluation as regular call expressions; first we evaluate the operator and then we evaluate the operands. That means that `woof` was printed when we evaluated the operand `(print 'woof)`. Inside the body of `twice`, the name `f` is bound to the value `undefined`, so the expression `(begin f f)` does nothing at all!

We have a problem here: we need to prevent the given expression from evaluating until we're inside the body of the procedure. This is where the `define-macro` special form, which has identical syntax to the regular `define` form, comes in:

```
scm> (define-macro (twice f) (list 'begin f f))
twice
```

`define-macro` allows us to define what's known as a **macro**, which is simply a way for us to combine unevaluated input expressions together into another expression. When we call macros, the operands are not evaluated, but rather are treated as Scheme data. This means that any operands that are call expressions or special form expression are treated like lists.

If we call `(twice (print 'woof))`, `f` will actually be bound to the list `(print 'woof)` instead of the value `undefined`. Inside the body of `define-macro`, we can insert these expressions into a larger Scheme expression. In our case, we would want a `begin` expression that looks like the following:

```
(begin (print 'woof) (print 'woof))
```

As Scheme data, this expression is really just a list containing three elements: `begin` and `(print 'woof)` twice, which is exactly what `(list 'begin f f)` returns. Now, when we call `twice`, this list is evaluated as an expression and `(print 'woof)` is evaluated twice.

```
scm> (twice (print 'woof))
woof
woof
```

To recap, macros are called similarly to regular procedures, but the rules for evaluating them are different. We evaluated lambda procedures in the following way:

1. Evaluate operator
2. Evaluate operands
3. Apply operator to operands, evaluating the body of the procedure

However, the rules for evaluating calls to macro procedures are:

1. Evaluate operator
2. Apply operator to unevaluated operands
3. Evaluate the expression returned by the macro **in the frame it was called in**.

Questions

Q5: Shapeshifting Macros (Tutorial)

When writing macros in Scheme, we often create a list of symbols that evaluates to a desired Scheme expression. In this question, we'll practice different methods of creating such Scheme lists.

We have executed the following code to define `x` and `y` in our current environment.

```
(define x '(+ 1 1))
(define y '(+ 2 3))
```

We want to use `x` and `y` to build a list that represents the following expression:

```
(begin (+ 1 1) (+ 2 3))
```

What would be the result of calling `eval` on a quoted version of the expression above?

```
(eval '(begin (+ 1 1) (+ 2 3)))
```

Now that we know what this expression should evaluate to, let's build our scheme list.

How would we construct the scheme list for the expression `(begin (+ 1 1) (+ 2 3))` using quasiquotation?

How would we construct this scheme list using the `list` special form?

How would we construct this scheme list using the `cons` special form?

Q6: Max Macro (Tutorial)

Define the macro `max`, which takes in two expressions `expr1` and `expr2` and returns the maximum of their values. If they have the same value, return the first expression. **For this question, it's okay if your solution evaluates `expr1` and `expr2` more than once.** As an extra challenge, think about how you could use the `let` special form to ensure that `expr1` and `expr2` are evaluated only once.

```
scm> (max 5 10)
10
scm> (max 12 12)
12
scm> (max 100 99)
100
```

First, try using quasiquotation to implement this macro procedure.

```
(define-macro (max expr1 expr2)
    'YOUR-CODE-HERE
    )

```

Now, try writing this macro using the `list` special form.

```
(define-macro (max expr1 expr2)
    'YOUR-CODE-HERE
    )

```

Finally, write this macro using the `cons` special form.

```
(define-macro (max expr1 expr2)
    'YOUR-CODE-HERE
    )

```

Q7: When Macro (Tutorial)

Using macros, let's make a new special form, `when`, that has the following structure:

```
(when <condition>
      (<expr1> <expr2> <expr3> ...))
```

If the condition is not false (a truthy expression), all the subsequent operands are evaluated in order and the value of the last expression is returned. Otherwise, the entire `when` expression evaluates to `okay`.

```
scm> (when (= 1 0) ((/ 1 0) 'error))
okay

scm> (when (= 1 1) ((print 6) (print 1) 'a))
6
1
a
```

```
(define-macro (when condition exprs)
  'YOUR-CODE-HERE
  )

```