Exercises

Homoiconicity

Quote exercises

Exercise: Find the value of each of the following in the repl:

```
+
(quote +)
(+ 1 2)
(quote (+ 1 2))
(= (quote +)
   (first (quote (+ 1 2))))
(= (+ 1 2)
   (quote (+ 1 2)))
(= (quote (+ 1 2))
   (list (quote +) 1 2))
(= ["string" :keyword #{:set} {:map true}]
     (quote ["string" :keyword #{:set} {:map true}]))
(quote (quote +))
```

Eval exercises

Exercise: Find the value of each of the following in the repl:

```
+
'+
(eval '+)

'(+ 1 2)
(eval '(+ 1 2))

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

Funhouse

Let’s build a fun-house alternate arithmetic universe where up is down and left is right. Addition becomes subtraction and multiplication is swapped with division.

1. `funhouse`

Define the function `funhouse`, so that the following work:

```
(funhouse '(+ 3 1)) ;=> (- 3 1)
(funhouse '(- 3 1)) ;=> (+ 3 1)
(funhouse '(* 3 1)) ;=> (/ 3 1)
(funhouse '(/ 3 1)) ;=> (* 3 1)
```

`funhouse` should take a form and return a form. It **should not** `eval` its results.

It should operate on the *symbols* `+`, `-`, `*`, `/`, not the *functions*. If your outputs contain #< in them, you have symbols that need to be quoted.

Once all that is sorted, you should be able to `eval` your function's return value and get the right result.

Useful functions: `first`, `rest`, `conj`, `quote`.

2. Recursive `funhouse`

Now, let's make `funhouse` work on more sophisticated examples. Make it work on trees of expressions.

Write a recursive version of `funhouse`. It should work on flat function calls, as in the previous exercise, but also arguments:

```
> (funhouse '(/ 3 (+ 2 1)))
(* 3 (- 2 1))
```

Start with this implementation skeleton:

```
(defn funhouse [expr]
  (if (list? expr)
    ...
    ...))
```

Useful functions: `map`

3. Extra credit

Expand `funhouse` to properly recurse into maps, vectors, and sets:

```
> (funhouse '(+ (/ 2 2)
              (first #{(- 1 1)})
              (first [(- 1 1)])
              (:2nd {:1st 1, :2nd (+ 5 1)})))
(- (* 2 2)
   (first #{(+ 1 1)})
   (first [(+ 1 1)])
   (:2nd {:1st 1, :2nd (- 5 1)}))
```

Macros

1. `in-funhouse`

`funhouse` is too inconvenient to use. It would be nice to have a macro `in-funhouse` that can be used without quoting. Create an `in-funhouse` macro that performs the `funhouse` transformation.

When you use it, it should work like this:

```
>(in-funhouse (+ 6 1))
5
```

When you `macroexpand-1` it, it should work like this:

```
>(macroexpand-1 '(in-funhouse (+ 6 1)))
(- 6 1)
```

Start with this definition:

```
(defmacro in-funhouse [expr]
…)
```

Hints: Reuse `funhouse`. **Do not** use `eval` in the implementation of this or any other macro.

2. `do-funhouse`

Let's take this a step further. Write a `do-funhouse` macro that can take multiple expressions. Here's an example of how we'd like to be able to use it:

```
>(do-funhouse (println (+ 1 2))
             (* 2 2)))
-1
1
```

Here's an example of how it should `macroexpand-1`:

```
> (macroexpand-1
    '(do-funhouse (println (+ 1 2))
                  (* 2 2))))
(do (println (- 1 2)) (/ 2 2))
```

To take arbitrarily many arguments, `do-funhouse` will need to be a variadic macro (taking arbitrarily many arguments). Here’s a start of the definition:

```
(defmacro do-funhouse [& exprs]
  …)
```

Functions you will need: `map`.

Syntax-quote

1. Rewrite debug using syntax-quote

Define debug using syntax-quotation to fix the variable capture problem. You will need to use syntax-quote (backtick) and ~ (unquote).

Your new debug implementation should work as follows:

```
> (macroexpand '(debug (+ 1 2)))
(do (println '(+ 1 2) :> (+ 1 2))
  (+ 1 2))
```

Keep in mind that

```
(list 'foo bar)
```

is roughly equivalent to the syntax-quoted

```
`(foo ~bar)
```

You should not need to call `list` anywhere in your solution.

Multiple execution

1. Fix multiple exection in debug

Our debug macro from earlier has another bug. Though we've fixed the variable capture problem with syntax-quote, we have another issue: our expressions are evaluated twice.

Consider this example:

```
> (debug (do (println "expression evaluated!") 1))
expression evaluated!
(do (println expression evaluated!) 1) :> 1
expression evaluated!
1
```

Fix it. Change the implementation of `debug` so that its argument is evaluated only once.

2. `and-1`

One core use of macros is control flow constructs. `and`, `or`, `when`, `cond`, `condp` are all macros. They have to be macros and not functions because their arguments are not always executed. Deciding if and when execution of arguments happens is the domain of macros.

Let's build up a 2-argument equivalent to `and` using macros. Recall that `and` short-circuits execution. For example

```
> (and nil (println "I will not print"))
nil
```

If the first argument is logically false, it is returned and the second argument is not executed. However, if the first argument is true, the second is executed and its value is returned.

```
> (and true :foo)
:foo
```

Write the macro `and-1` which takes exactly two arguments and is equivalent to `and`.

Also note that `and-1` should evaluate arguments at most once, as shown in the following example:

```
> (and-1 (do (println "I will print")
             true)
         1)
I will print
1
```

Using functions in macros

Practice compile-time “optimization” with a simplified example of the type of work hiccup is doing. Let’s create a macro `pre-str` which looks for static elements and converts them to a string at compile time. (This macro is unnecessary and doesn’t really optimize anything, but is good practice.)

We’ll be creating this macro in stages. In the end, `pre-str` should work just like `str` in terms of runtime behavior:

```
> (pre-str 1 " " :foo " " (+ 1 2) " " *file*)
"1 :foo 3 NO_SOURCE_PATH"
```

However, `pre-str` converts static values into strings:

```
> (macroexpand-1 '(pre-str 1 " " :foo " " (+ 1 2) " " *file*))
(clojure.core/str "1" " " ":foo" " " (+ 1 2) " " *file*)
```

`pre-str` will detect static values and convert them into strings. Run-time values should still be executed.

1. `stringify-static`

Define the function `stringify-static` which takes a sequence of values and returns a sequence with all static values converted to strings.

```
> (stringify-static '(:foo 1 (+ 1 2) "jon" bar))
(":foo" "1" (+ 1 2) "jon" bar)
```

Helpful Clojure: `keyword?`, `string?`, `number?`, `map`, `or`, `some-fn`.

2. `pre-str`

Create the macro `pre-str` which works as described above. Call `stringify-static` from it in order to accomplish part of the work.

3. Extra credit

Modify `stringify-static` to convert runs of static values into a single string. The updated `pre-str` implementation should expand as follows:

```
> (macroexpand-1 '(pre-str 1 " " :foo " " (+ 1 2) " " *file*))
(clojure.core/str "1 :foo " (+ 1 2) " " *file*)
```

Helpful Clojure: `apply`, `partition-by`, and possibly `mapcat`.

Recursive macros

1. and-*

Define and-*, which works just like and. This should be a recursive macro which can take an arbitrary number of arguments instead of only two. See the example of -> below for ideas.

Your macro will need to be recursive in order for the following cases to work:

```
> (and-* 0)
0
> (and-* 0 1)
1
> (and-* 0 1 2)
2
> (and-* 0 nil (println "I will not print"))
nil
```

Hint: (and 1 2 3) is equivalent to (and 1 (and 2 3)).
Hint 2: Your macro should not recurse directly, but emit a call to itself.

The definition of ->, an example recursive macro:

```
(defmacro ->
  ([x] x)
  ([x form] (if (seq? form)
              `(~(first form) ~x ~@(next form))
              (list form x)))
  ([x form & more] `(-> (-> ~x ~form) ~@more)))
```

Proceed to handout

Start from page 39 in the handout for more material and exercises.