Class Exercises: Macros

1. Write a Clojure macro called `bif` (*bound if*) that has the following syntax:

   ```
   (bif variable condition-part
     then-part
     else-part)
   ```

   This macro evaluates *condition-part*, binds the result to *variable* and, if it's a truthy value, evaluates and returns *then-part*, otherwise evaluates and returns *else-part*. The scope of *variable* includes *then-part* and *else-part*. In other words, the `bif` macro expands to the following code:

   ```
   (let [variable condition-part]
     (if variable
       then-part
       else-part))
   ```

   Examples:

   ```
   (macroexpand-1
     '(bif q (first '(4 8 15))
        (inc q)
        (list q)))
   => (clojure.core/let [q (first (quote (4 8 15)))] (if q (inc q) (list q)))

   (bif q (first '(4 8 15))
     (inc q)
     (list q))
   => 5

   (bif q (first ())
     (inc q)
     (list q))
   => (nil)
   ```

2. Write a Clojure macro called `def-vars` that receives a symbol *var-name* and zero or more expressions. This macro defines as many global variables as the provided number of expressions. The value of *var-name* is the name prefix for all these variables. The suffix is "0" for the first variable name, which is initialized with the first expression. The next variable name has a "1" suffix and is initialized with the second expression, and so on with all the remaining variables.

   For example, the expression:

   ```
   (def-vars x (+ 1 2) 3 (* 2 2))
   ```

   should macroexpand to:

   ```
   (do
     (def x0 (+ 1 2))
     (def x1 3)
     (def x2 (* 2 2)))
   ```

   When the macro is evaluated, variables `x0`, `x1` and `x2` should be defined. Thus:

   ```
   (+ x0 x1 x2)
   => 10
   ```

   These are some functions you might find usefull:

   ```
   (str 'foo 123)
   => "foo123"

   (symbol "foo123")
   => foo123
   ```

3. Write a Clojure macro called `def-many`. This macro allows defining many global bindings in one place. It has the following form:

   `(def-many` var_1 $expr_1$ var_2 $expr_2$ ... var_n $expr_n$`)`

   Where every var_i is a symbol and every $expr_i$ is an arbitrary expression. The macro evaluates $expr_1$ and binds the result to var_1 (using the `def` special form), then evaluates $expr_2$ and binds the result to var_2, and so on.

   The macro expands to the following form:

   ```
   (do
     (def var_1 expr_1)
     (def var_2 expr_2)
           ⋮
     (def var_n expr_n))
   ```

   Examples:

   ```
   (macroexpand-1 '(def-many a (+ 1 2)
                             b (* 2 a)
                             c (/ (inc b) a)))
   => (do
        (def a (+ 1 2))
        (def b (* 2 a))
        (def c (/ (inc b) a)))

   (def-many a (+ 1 2)
             b (* 2 a)
             c (/ (inc b) a))
   (+ a b c)
   => 34/3
   ```

4. Write a Clojure macro called `nth-expr`. This macro only evaluates the n-th item of a series of expressions. It has the following form:

   (`nth-expr` *nth* $expr_0$ $expr_1$ ... $expr_k$)

   The macro evaluates the expression *nth*, if it's equal to 0 it returns the result of evaluating $expr_0$, otherwise, if it's equal to 1 it returns the result of evaluating $expr_1$, and so on. Only one of $expr_0$, $expr_1$, ..., $expr_k$ is actually evaluated, the rest of the expressions are ignored. A runtime exception is thrown if the result of evaluating *nth* is not an integer between 0 and k.

   The macro expands to the following `case` form:

   (`case` *nth*
   `0` $expr_0$
   `1` $expr_1$
   ⋮
   k $expr_k$
   `(throw (RuntimeException. "Bad nth value!"))`)

   Examples:

```
(macroexpand-1 '(nth-expr (- 5 4) (* 2 3) (- 5 2) (+ 7 2) (/ 20 2)))
=> (clojure.core/case (- 5 4)
     0 (* 2 3)
     1 (- 5 2)
     2 (+ 7 2)
     3 (/ 20 2)
     (throw (java.lang.RuntimeException. "Bad nth value!")))

(nth-expr (- 5 4) (* 2 3) (- 5 2) (+ 7 2) (/ 20 2))
=> 3

(nth-expr :wat 0 1 2 3 4 5)
=> RuntimeException Bad nth value!
```