As with all Lisps, Clojure's inherent homoiconicity gives you access to the full extent of the language to write code-generation routines called "macros". Macros provide a powerful way to tailor the language to your needs.

Be careful though. It's considered bad form to write a macro when a function will do. Use a macro only when you need control over when or if the arguments to a form will be evaluated.

You'll want to be familiar with Clojure. Make sure you understand everything in Clojure in Y Minutes.

```
;; Define a macro using defmacro. Your macro should output a list that can
;; be evaluated as clojure code.
;;
;; This macro is the same as if you wrote (reverse "Hello World")
(defmacro my-first-macro []
  (list reverse "Hello World"))

;; Inspect the result of a macro using macroexpand or macroexpand-1.
;;
;; Note that the call must be quoted.
(macroexpand '(my-first-macro))
;; -> (#<core$reverse clojure.core$reverse@xxxxxxxx> "Hello World")

;; You can eval the result of macroexpand directly:
(eval (macroexpand '(my-first-macro)))
; -> (\d \l \o \r \W \space \o \l \l \e \H)

;; But you should use this more succinct, function-like syntax:
(my-first-macro)  ; -> (\d \l \o \r \W \space \o \l \l \e \H)

;; You can make things easier on yourself by using the more succinct quote syntax
;; to create lists in your macros:
(defmacro my-first-quoted-macro []
  '(reverse "Hello World"))

(macroexpand '(my-first-quoted-macro))
;; -> (reverse "Hello World")
;; Notice that reverse is no longer function object, but a symbol.

;; Macros can take arguments.
(defmacro inc2 [arg]
  (list + 2 arg))

(inc2 2) ; -> 4

;; But, if you try to do this with a quoted list, you'll get an error, because
;; the argument will be quoted too. To get around this, clojure provides a
;; way of quoting macros: `. Inside `, you can use ~ to get at the outer scope
(defmacro inc2-quoted [arg]
  `(+ 2 ~arg))

(inc2-quoted 2)

;; You can use the usual destructuring args. Expand list variables using ~@
(defmacro unless [arg & body]
  `(if (not ~arg)
```

```
     (do ~@body))) ; Remember the do!

(macroexpand '(unless true (reverse "Hello World")))
;; ->
;; (if (clojure.core/not true) (do (reverse "Hello World")))

;; (unless) evaluates and returns its body if the first argument is false.
;; Otherwise, it returns nil

(unless true "Hello") ; -> nil
(unless false "Hello") ; -> "Hello"

;; Used without care, macros can do great evil by clobbering your vars
(defmacro define-x []
  '(do
     (def x 2)
     (list x)))

(def x 4)
(define-x) ; -> (2)
(list x) ; -> (2)

;; To avoid this, use gensym to get a unique identifier
(gensym 'x) ; -> x1281 (or some such thing)

(defmacro define-x-safely []
  (let [sym (gensym 'x)]
    `(do
       (def ~sym 2)
       (list ~sym))))

(def x 4)
(define-x-safely) ; -> (2)
(list x) ; -> (4)

;; You can use # within ` to produce a gensym for each symbol automatically
(defmacro define-x-hygenically []
  `(do
     (def x# 2)
     (list x#)))

(def x 4)
(define-x-hygenically) ; -> (2)
(list x) ; -> (4)

;; It's typical to use helper functions with macros. Let's create a few to
;; help us support a (dumb) inline arithmetic syntax
(declare inline-2-helper)
(defn clean-arg [arg]
  (if (seq? arg)
    (inline-2-helper arg)
    arg))

(defn apply-arg
```

```
  "Given args [x (+ y)], return (+ x y)"
  [val [op arg]]
  (list op val (clean-arg arg)))

(defn inline-2-helper
  [[arg1 & ops-and-args]]
  (let [ops (partition 2 ops-and-args)]
    (reduce apply-arg (clean-arg arg1) ops)))

;; We can test it immediately, without creating a macro
(inline-2-helper '(a + (b - 2) - (c * 5))) ; -> (- (+ a (- b 2)) (* c 5))

; However, we'll need to make it a macro if we want it to be run at compile time
(defmacro inline-2 [form]
  (inline-2-helper form)))

(macroexpand '(inline-2 (1 + (3 / 2) - (1 / 2) + 1)))
; -> (+ (- (+ 1 (/ 3 2)) (/ 1 2)) 1)

(inline-2 (1 + (3 / 2) - (1 / 2) + 1))
; -> 3 (actually, 3N, since the number got cast to a rational fraction with /)
```

Further Reading

Writing Macros from Clojure for the Brave and True http://www.braveclojure.com/writing-macros/

Official docs http://clojure.org/macros

When to use macros? http://dunsmor.com/lisp/onlisp/onlisp_12.html