U11BV7MTK : <https://github.com/clojure-emacs/cljs-tooling/blob/master/src/cljs_tooling/util/misc.clj#L7>
U11BV7MTK : it's a `cond-let` macro
U08QZ7Y5S : Thanks <@U11BV7MTK>, I'll check that out. Nothing built-in I'm missing then, I take it?
U11BV7MTK : not as far as i know
U08QZ7Y5S : Cool, thanks. Looks like that macro exists verbatim in a lot of libraries...
<https://crossclj.info/clojure/cond-let.html>
U11BV7MTK : the exact same? I guess its just one of those clojure archetypes
U0QNQ3P3L : I am curious as to where people are storing simple jdbc queries? I have a set of ~8 or so queries, all plain queries without any parameters. Do you use a "config" type file?
U065JNAN8 : Stick 'em in the resources directory of your project then you can retrieve them by slurping the return value of `(<http://clojure.java.io/resource|clojure.java.io/resource> "foo.sql")`
U0QNQ3P3L : <@U065JNAN8> - yes, certainly in the resources directory. I was just wondering if there was a preferred file format but a .sql file makes the intent of the file very apparent, which is good.
U0QNQ3P3L : Thanks!
U1ACUMJKX : hey I was thinking about a way to minimize the memory requirements of a nested clojure data structure where some substructures are identical, and I came up with this: ```(partial clojure.walk/postwalk (memoize identity))``` but I haven't really given it much thought. How would you do something like this?
U051SS2EU : <@U1ACUMJKX> if you just use the same object as an arg to assoc, it won't be duplicated
U051SS2EU : depending on how the data was created, of course
U1ACUMJKX : my use case involves taking an .edn file from disk that is probably too self similar and large to fit in memory
U051SS2EU : oh, yeah, fun
U051SS2EU : it would be interesting to try the postwalk identity and then compare the object pointers via jdb maybe(?)
U1ACUMJKX : yeah i haven't tested it i was just wondering if anyone else had ideas
U65U08BB4 : what's "fn*"? I cannot find the document of it~
U051SS2EU : it's an implementation detail of fn
U051SS2EU : fn is implemented as a macro, and uses the destructuring functions that clojure.core defines for macros
U65U08BB4 : so means: I don't have to care about it?
U051SS2EU : fn* is implemented in java code
U051SS2EU : right, remembering that it's an fn that can't destructure is probably enough
U65U08BB4 : could you please give an example? of the difference?
U051SS2EU : ```+user=> ((fn [[a]] a) [1])1
+user=> ((fn* [[a]] a) [1])
CompilerException java.lang.IllegalArgumentException: fn params must be Symbols,
compiling:(NO_SOURCE_PATH:2:2)```

U051SS2EU : [a] as a parameter is a destructure that says "bind the first element of this sequencable input to the name a"
U051SS2EU : fn* doesn't understand that syntax
U65U08BB4 : so the difference is only about the destructuring of the parameters? fn supports it, while fn* doesn't?
U051SS2EU : that's the main one, I forget if it's the only one
U65U08BB4 : hmm~ in lazy-seq macro:
U65U08BB4 : boot.user=> (source lazy-seq)(defmacro lazy-seq
  "Takes a body of expressions that returns an ISeq or nil, and yields
  a Seqable object that will invoke the body only the first time seq
  is called, and will cache the result and return it on all subsequent
  seq calls. See also - realized?"
  {:added "1.0"}
  [& body]
  (list 'new 'clojure.lang.LazySeq (list* '^{:once true} fn* [] body)))

U65U08BB4 : ```boot.user=> (source lazy-seq)(defmacro lazy-seq
  "Takes a body of expressions that returns an ISeq or nil, and yields
  a Seqable object that will invoke the body only the first time seq
  is called, and will cache the result and return it on all subsequent
  seq calls. See also - realized?"
  {:added "1.0"}
  [& body]
  (list 'new 'clojure.lang.LazySeq (list* '^{:once true} fn* [] body)))```

U65U08BB4 : why is fn* preferred here~? is it some performance concern, as fn* is the basic one?
U051SS2EU : right, destructuring is defined in terms of lazy-seq