Heinrich-Heine-University Düsseldorf
Computer Science Department
Software Engineering and Programming Languages
Philipp Körner

Functional Programming – WT 2023 / 2024
Reading Guide 7: Macros

Timeline: This unit should be completed by 27.11.2023.

1 Material

- 17_homoiconicity.clj
- 11_macros.clj
- Clojure for the Brave and True, chapter 8 (contains next week's material as well)
- Clojure Reference: Macros `https://clojure.org/reference/macros`
- Learning Video: Macros: `https://mediathek.hhu.de/watch/7e4ba161-f91b-4c7a-a1e2-e135c861865c`

2 Learning Outcomes

After completing this unit you should be able to

- fully describe the process of evaluation of expressions in Clojure.
- define and identify homoiconicity.
- write macros by yourself.

3 Highlights

- Macroexpension
- Complete evaluation rules
- Helpers: `macroexpand`, `macroexpand-1`, `macroexpand-all`
- Macro: `defmacro`

4 Exercises

Note: the next reading guide will contain the same exercises. You may — for didactical purposes — try these exercises now to experience the pain of writing macros without the templating syntax.

Exercise 7.1 (Macro)

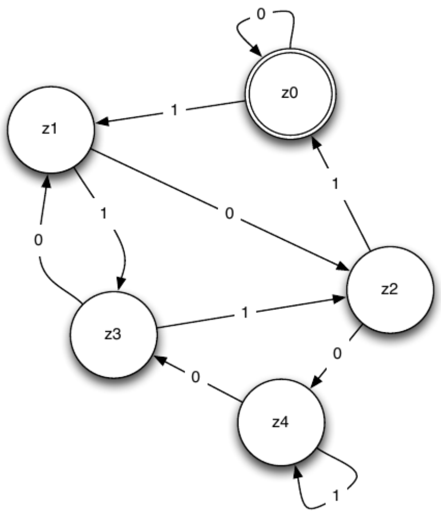
a) Write a macro that implements a simple calculator with infix notation. The first argument should be a vector with bindings of symbols to values. Parentheses and operator precedences need not be considered. The macro should work as follows:

```
user=> (calc [a 3] 3 + 5 * a)
24
user=> (calc [b 5 a 3] a * b + 3)
18
```

b) Why can you not write this calculator as a function?

Exercise 7.2 (DFA)

Your task is to develop a simple DSL for DFAs. The automaton that recognizes binary numbers divisible by five will serve as example. The states in the automaton are labeled according to the residue class modulo 5, so z0 represents all numbers divisible by 5, z3 all numbers whose remainder is 3, etc.



The DSL should look as follows (i.e., *this exact code is expected to work without any modification!* It may only be executed once, so once you have a solution, re-start your REPL to ensure you do not have any left-over state.):

```
(declare z0 z1 z2 z3 z4)
(def z0 (dfa-state :accept {\0 z0 \1 z1}))
(def z1 (dfa-state :reject {\0 z2 \1 z3}))
(def z2 (dfa-state :reject {\0 z4 \1 z0}))
(def z3 (dfa-state :reject {\0 z1 \1 z2}))
(def z4 (dfa-state :reject {\0 z3 \1 z4}))
```

Nota bene: You have to use the code snippet above verbatim! No modifications are allowed.
`(dfa-state ... )` should return a function, which receives a string, consisting of 0s and 1s, as parameter and returns :accept, if the string starting from this node results in an accepting state. Otherwise :reject should be returned.

`(dfa-match init text)` should serve as entry point, which, starting from init, processes the input text.

```
user=> (dfa-match z0 "")
:accept
user=> (dfa-match z0 "010100010")
:reject
user=> (dfa-match z0 "0101000101")
:accept
```

Test your implementation with larger inputs (i.e. longer strings).

Note: It is likely that your first implementation will not work. Do not let yourself be discouraged by this and try to locate the problem, in order to fix it afterwards.

Questions

If you have any questions, please contact Philipp Körner (`p.koerner@hhu.de`) or post it to the Rocket.Chat group.