1.3.4 case and case*

macro since 1.2

Listing 3. 14. → Conditional Branching, Fast Switch

```
(case [expression & clauses])
```

`case` is a conditional statement which accepts a list of testing conditions to determine which branch to evaluate. `case` is not much different from [cond] or [condp] and it can be considered part of the same family of macros:

```
(let [n 1]
  (case n
    0 "O"
    1 "l"
    4 "A"))

;; l
```

`case` is perhaps unique for its treatment of case tests constants that are not evaluated at macro-expansion time. This means that a form like `(inc 0)` is not evaluated and replaced as `1` when appearing as a test expression. It is instead considered the group containing the symbol `inc` and the number 0 as literal constants:

```
(let [n 1]
  (case n
    (inc 0) "inc" ; ❶
    (dec 1) "dec" ; ❸
    :none))

;; "dec"
```

❶ To be read as: "is n=1 present in the set formed by the symbol 'inc and the number 0? No.

➋ Similarly: "is n=1 present in the set formed by the symbol 'dec and the number 1? Yes.

`case*` is one of the few special forms in Clojure. The only use in the standard library is wrapped by the `case` macro which is the main entry point for `case*`. There is no particular value in using it directly, so all information about it can be found in this chapter. Compared to other conditional forms, `case` is specifically designed with performances in mind. `case` implementation compiles into the optimized `tableswitch` JVM bytecode instruction [14] providing constant time lookup (instead of linear as in [cond]) at the cost of some restrictions around the test expressions.

Listing 3. 15. Contract

```
  (case <expr> [clauses] [<default>])
```

[14] To know more about `tableswitch` JVM instruction please read the following article about control flow in the Java virtual machine: http://www.artima.com/underthehood/flowP.html

```
clause :=> <test> <then>
```

INPUT

- "expr" is mandatory and can be any valid Clojure expression.
- "clauses" are grouped into one or more pairs. If there are no clauses, there should be at least one "default".
- "pair" is a "test" followed by a "then".
- "test" is a compile-time literal and is not evaluated at macro-expansion time. Valid literals are: `:a` (keywords), `'a`(symbols), `1`, `1.0`, `1M`, `1N` (numbers), `{} #{} () []` (collection literals with a specific meaning for lists), "" (strings), `\a` (chars) and `1/2` (ratios).
- "then" is any valid Clojure form or expression that will be evaluated when the corresponding "test" is matching.
- "default" is any valid Clojure form or expression.

EXCEPTIONS

`java.lang.IllegalArgumetnException` when:

- There is no matching "test" for the given expression and no "default" is given.
- There is a duplicate "test" constant.

OUTPUT

- returns the "default" if one or more clauses are present but none is matching.
- returns the evaluation of "then" for the first pair-clause where `(identical? test expr)` is `true`.

Examples

Let's first clarify some aspects of the contract. `case` tests are considered compile time literals with implications like the following:

```
(case 'pi 'alpha \α 'beta \β 'pi \π) ; ❶
;; IllegalArgumentException: Duplicate case test constant: quote

(macroexpand ''alpha) ; ❸
;; (quote alpha)

;; (case 'pi (quote alpha) \α (quote beta) \β (quote pi) \π) ; ❹

(case 'pi alpha \α beta \β pi \π) ; ❹
;; \π
```

❶ Symbols like `'alpha` that would otherwise be evaluated as the symbol itself in normal forms, are not interpreted as such here.

❷ `case` considers the quoted version of `'alpha` as a test, resulting in something like this macroexpansion.

❸ This is what `case` actually sees. Since parenthesis interpretation is similar to normal Clojure set, the symbol "quote" is appearing in each of the tests, resulting in the exception because `case` doesn't know what to pick in case the input is "quote".

❹ The correct way to match against the symbols is to completely remove the single quote ad the beginning.

To avoid surprises, please take particular care whenever `case` is used with special constant literals (other than the commonly used numbers, strings or chars). At the same time we can take advantage of the special meaning for lists to match against multiple inputs at once, like for the following infix calculator:

```
(defn error [& args]
  (println "Unrecognized operator."))

(defn operator [op]                                          ; ❶
  (case op
    ("+" "plus") +
    ("-" "minus")
    ("*" "x" "·" "times") *
    ("/" "÷") /
    error))

(defn execute [arg1 op arg2]                                 ; ❸
  (apply
    (operator op)
    (map #(Integer/valueOf %) [arg1 arg2])))

(defn calculator [s]                                         ; ❹
  (let [[arg1 op arg2] (clojure.string/split s #"\s")]
    (execute arg1 op arg2)))

(calculator "10 ÷ 5")                                        ; ❹
;; 2
```

❶ `operator` translates an operator as string into the corresponding Clojure function. We can use `case` to select between the fundamental operations or an `error` function to handle unrecognized operators. Note how we can add multiple synonyms of the 4 basic operations as list for `case` to evaluate as inclusion.

❷ `execute` is our simple [eval] equivalent that takes operators and operand and execute the corresponding operation once it has been translated.

❸ `calculator` is the main entry point. It is roughly equivalent to the parsing phase of a compiler, since it takes the raw unevaluated string and converts it into "tokens" (our abstract syntax tree) ready for evaluation.

❹ Invoking the calculator produces the expected results.

Since lists have a special meaning for `case`, we are apparently in trouble if we wanted to match against actual lists. How to to use actual lists as tests in that case? Vectors can be used to match against lists. The following examples shows basic way to score a Vim user

[15] based on the best key combination to achieve some editing task (usually the shortest amount of keystrokes wins). For simplicity we are going to consider the very simple task of moving the cursor from the lower-left corner of a 5x5 grid to the upper right corner, like shown in the picture below:

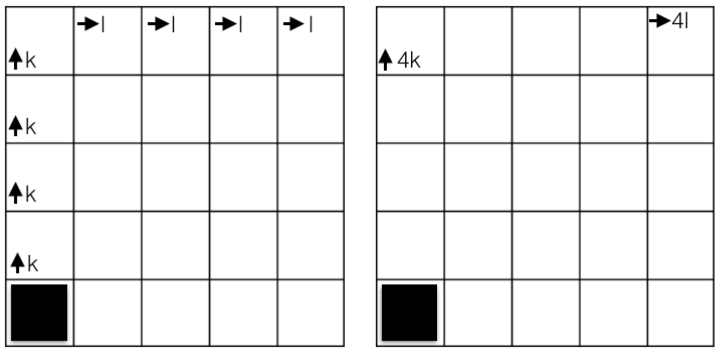

Figure 3.2 Visually representing Vim keystrokes movement to move from one corner to the other.

The letter "k" moves up the cursor while the letter "l" moves it to the right. One poor solution would be to hit "k" 4 times followed by hitting "l" 4 more times (diagram on the left): in this case we are going to acknowledge the accomplishment but giving a low score. A better solution would be to press "4" followed by the moving letter, reducing the number of keystroke to a half compared to the previous solution (picture on the right). The code to score such a result could be implemented as:

```
(defn score [ks]
  (case ks
    [\k \k \k \k \l \l \l \l] 5    ; ❶
    [\4 \k \4 \l] 10
    0))

(defn check [s]
  (score (seq s)))                 ; ❸

(check "kl")
;; 0

(check "kkkkllll")
;; 5

(check "4k4l")
;; 10
```

[15] Vim is a popular text editor that thanks to editing contexts has very short key combinations. See https://en.wikipedia.org/wiki/Vim_(text_editor) to know more.

❶ We are grouping the char constants in a vector, each vector representing one test for the `case` statement. `case` does not consider this a duplication of the test expression.
❷ Since the input is a string, we just need to call [seq] on it to transforms it into a sequence of characters.

There are a couple of things to note about this example:

- There is implicit ordering of the keystrokes determined by the [vector] ordering. Clojure [set] could be used as a constant container for those cases where ordering doesn't matter.
- `score` is invoked passing a sequence as an argument. `case` doesn't distinguish between lazy-sequences and vectors, so the two are interchangeable and can be used as literals.

"case" and table branching

`case` is implemented similarly to a well known compiler optimization used in switch statements (also called "case" or "select" statements).

The idea is the following: transform the tests constants into keys suitable for hashing and use hash-table lookup to check if there is a match. The problem then translates to transforming constants into integers. There is also another important aspect to consider: if the keys are contiguous (that is, no gaps between consecutive integers) then it's possible to enter the switch based on a simple condition to check if the expression is or not in the allowed range. Clojure has the advantage that the JVM already provides some abstraction to build the lookup table with the `tableswitch` opcode which requires the following:

- The test values to be `int` or int-equivalents (char, bytes, shorts).
- The test values to be contiguous (potentially adding the default case label as many times as needed in between to fill the gaps).
- The total size of the switch table shouldn't be more than 8192 bytes.

The practical implications for Clojure is that there must be a way to transform compile time constants or grouping thereof into integers and shift/mask the integers to obtain the smallest possible gap in between keys. Another potential problem happens on hash-collisions and in general when transforming composites into integers. So despite the simple idea, Clojure has to do quite a lot of non-trivial processing to get it right [16] . A few fairly complicated functions (`prep-hashes`, `merge-hash-collisions`, `fits-table?` and others) are dedicated in "core.clj" to transform `case` constants into a gap-less list of non-clashing integers.

See Also

- [cond] has a similar semantic compared to `case`. The most notable difference is the possibility to evaluate test expressions at compile time.
- [condp] allows to input the predicate that should be used for matching and adds the additional `:>>` semantic.

[16] A good selection of `case` corner cases is visible on this ticket: http://dev.clojure.org/jira/browse/CLJ-426

[cond] and [condp] are in general more flexible. As a rule of thumb, prefer `case` in the presence of literals or when performances are important.

Performance Considerations and Implementation Details

⇒ `O(n)` macro expansion time

⇒ `O(1)` runtime

The main selling point of `case` is the constant time access lookup independently from the number of test-then pairs present in the statement. We can quickly verify the claim using Criterium[17]:

```
(require '[criterium.core :refer :all])

(defn c1 [n]
 (cond
  (= n 0) "0" (= n 1) "1"
  (= n 2) "2" (= n 3) "3"
  (= n 4) "4" (= n 5) "5"
  (= n 6) "6" (= n 7) "7"
  (= n 8) "8" (= n 9) "9"
  :default :none))

(bench (c1 9))
;; Execution time mean : 10.825367 ns

(defn c2 [n]
 (case n
  0 "0" 1 "1"
  2 "2" 3 "3"
  4 "4" 5 "5"
  6 "6" 7 "7"
  8 "8" 9 "9"
  :default))

(bench (c2 9))
;; Execution time mean : 6.716657 ns
```

As you can see the mean execution time goes from 10.825367 ns for the version using [cond] to the 6.716657 ns for the version using `case` which is about 40% faster. The speedup is also given by the fact that [cond] is using the "=" equality operator while `case`, being based on constant literals, is implicitly using reference equality. A more "fair" benchmark could use identical?, but that would restrict the normal operational spectrum of [cond] with potentially surprising results:

```
(defn c1 [n]
  (case n 127 "127" 128 "128" :none))

(c1 127)
;; "127"
(c1 128)    ; ❶
```

[17] Criterium is the de-facto benchmariking tool for Clojure: https://github.com/hugoduncan/criterium

```
;; "128"

(defn c2 [n]
  (cond (identical? n 127) "127" (identical? n 128) "128" :else :none))

(c2 127)
;; "127"
(c2 128)    ; ❸
;; :none
```

❶ `case` correctly reports "128" as the correct answer

❷ [cond] with identical? doesn't enter the expected branch because of the internal JVM caching of boxed Integers only being available up to 127[18] .

Please note that there is nothing wrong with the implementation of [cond] but it has more to do with the implication of using identical? as the equality operator. `case` simply avoids the additional thinking required to understand the implications of using identical?.

If we macroexpand a simple example, we can see how `case` delegates down to `case*` special form passing down the arguments that are needed to create the necessary bytecode:

```
(macroexpand
  '(case a 0 "0" 1 "1" :default))

;; (let*
;;   [G__759 a]
;;     (case* G__759
;;       0 0 :default
;;       {0 [0 "0"], 1 [1 "1"]}
;;       :compact :int))
```

Going further down to the produced JVM bytecode, the `case*` special form produces the following (showing just the main `tableswitch` and related details):

```
(require '[no.disassemble :refer [disassemble]])
(println (no.disassemble/disassemble #(let [a 8] (case a 0 "0" 1 "1" :default))))

;; showing just the very beginning

     0  ldc2_w <Long 8> [12]
     3  lstore_1 [a]
     4  lload_1 [a]
     5  lstore_3 [G__22423]
     6  lload_3 [G__22423]
     7  l2i
     8  tableswitch default: 54
          case 0: 32
          case 1: 43
```

[18] See https://www.owasp.org/index.php/Java_gotchas#Immutable_Objects_.2F_Wrapper_Class_Cachingto know how Java internal caching of boxed values works

Thanks to the requirement that `case` can only process compile-time constants, the `tableswitch` instruction already contains all the necessary information to execute without the need to further evaluate other forms or vars.