How to read input from the keyboard, while also validating the data (CLIPS style)

```
(deffacts initial-phase
   (phase choose-player))

(defrule player-select
   (phase choose-player) ; control pattern
   =>
   (printout t "Who moves first (Computer: c "
               "Human: h)? ")
   (assert (player-select (read))))    ; reads c or h from the keyboard
                                       ; and asserts it in momory

(defrule good-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&c | h)  ; if the user has inputed c or h
   =>
   (retract ?phase ?choice)
   (assert (player-move ?player)))

(defrule bad-player-choice    ; if bad, then back to previous page
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&~c&~h)
   =>
   (retract ?phase ?choice)
   (assert (phase choose-player))
   (printout t "Choose c or h only." crlf)
   )
```

The example functions like this: first, in the working memory is the fact (phase chose-player). It will trigger the rule “player-select”. It will read the user’s input from the keyboard with the (read) function, placing it directly into the memory as a fact (player-select r), for exemplae, if the user inputed r as the answer from the keyboard.
Since r is not a good choice, we expect the user to input either c or h from the keyboard, it will trigger the rule bad-player-choice, because in memory is a fact matching (player-select ?player&~c|~h), which means “the value of the ?player variable is not c (~c) or (|) not h (~h), so it is neither c nor h”.
The facts whose addresses are ?phase and ?choice are retracted from the memory `(retract ?phase ?choice)`, instead we assert again the fact `(assert (phase choose-player))`
It might seem odd why we retract the existing fact `(phase choose-player)` and then we assert it again into memory. The answer is that on the new fact, the rule player-select will trigger again, which woldn’t happen on the old fact `(phase choose-player)`, already marked as used by the rule player-select. So we want to force the rule that reads the user input to trigger again, reading a new input, eventually a good one this time.
So we can se a way of doing a “while”-like iteration without using a while instruction, only be asserting and retracting facts into memory.

How to perform a while loop

```
CLIPS>
(deffunction iterate (?num)
(bind ?i 0)
(while TRUE do
(if (>= ?i ?num) then
(break))   ; the forced exit condition if ?i >= ?num
(printout t ?i " ")
(bind ?i (+ ?i 1)))   ; increase the cycle variable
(printout t crlf))
CLIPS> (iterate 1)
0
CLIPS> (iterate 10)
0 1 2 3 4 5 6 7 8 9
CLIPS>
```

Another example:

```
(defrule open-valves
(valves-open-through ?v)
=>
(while (> ?v 0)
(printout t "Valve " ?v " is open" crlf)
(bind ?v (- ?v 1))))
```

How to perform a “for” loop

```
CLIPS> (loop-for-count 2 (printout t "Hello world" crlf))
Hello world
Hello world
FALSE
CLIPS>
(loop-for-count (?cnt1 2 4) do  ; 2 and 4 designates the range of values for ?cnt1
  (loop-for-count (?cnt2 1 3) do
    (printout t ?cnt1 " " ?cnt2 crlf)))
2 1
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
FALSE
CLIPS>
```

How to use a if-then-else construct

```
(defrule closed-valves
(temp high)
(valve ?v closed)
=>
(if (= ?v 6)
then
(printout t "The special valve " ?v " is closed!" crlf)
(assert (perform special operation))
else
(printout t "Valve " ?v " is normally closed" crlf)))
```

Note that this rule could have been accomplished just as easily with two rules, and that it is usually better to accomplish this with two rules.

```
(defrule closed-valves-number-6
(temp high)
(valve 6 closed)
=>
(printout t "The special valve 6 is closed!" crlf)
(assert (perform special operation)))
(defrule closed-valves-other-than-6
(temp high)
(valve ?v&~6 closed)
=>
(printout t "Valve " ?v " is normally closed" crlf))
```

Reading and writing from files

```
(open "example.dat" xmp "w")
(printout xmp "green" crlf)
(printout xmp 7 crlf)
(close xmp)

(open "example.dat" xmp "r")
(read xmp)
(read xmp)
(close xmp)
```

The previous examples will open for writing a file named example.dat, assigning the file handle xmp to it.
Later we can use the file handle to write into the file, as with `(printout xmp "green" crlf)`
The writings will be done one element per line, since the read function used later only reads one element, not the entire line.
It is clear that we will use also the “printout xmp” constructs embedded in more complex constructs, mainly as a part of some rules.
Example:
(deffacts initial-facts
(start))

(defrule read-the-file
(start)
=>
`(open "example.dat" xmp "r")`
(bind ?x (read xmp))
(bind ?y (read xmp))
(printout t “There are “ ?y ?x “apples”)
`(close xmp)`

)

(defrule create-the-file

)