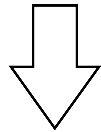
# CLIPS-03

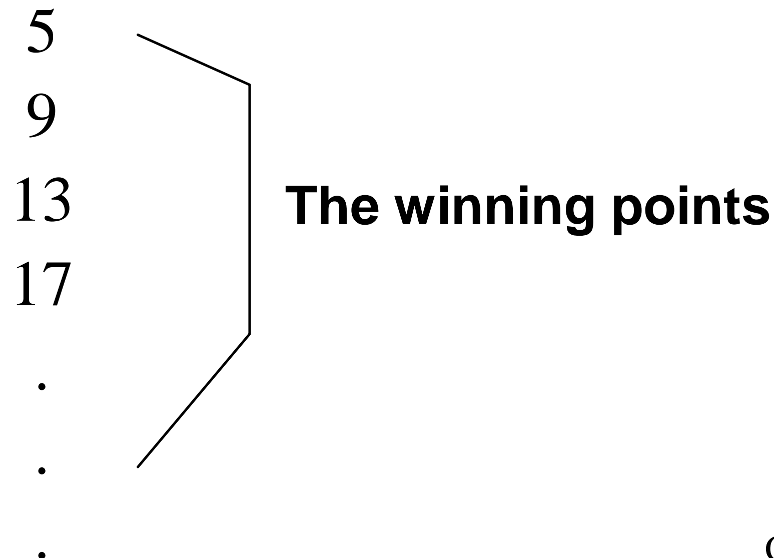
# Control Techniques

By Gwo-Jen Hwang

- Game of Sticks
  (poison.clp)



Initially, there are n sticks.
Two persons take the sticks in turn.
Each time only one to three sticks can be taken.
The one who takes the last stick loses.



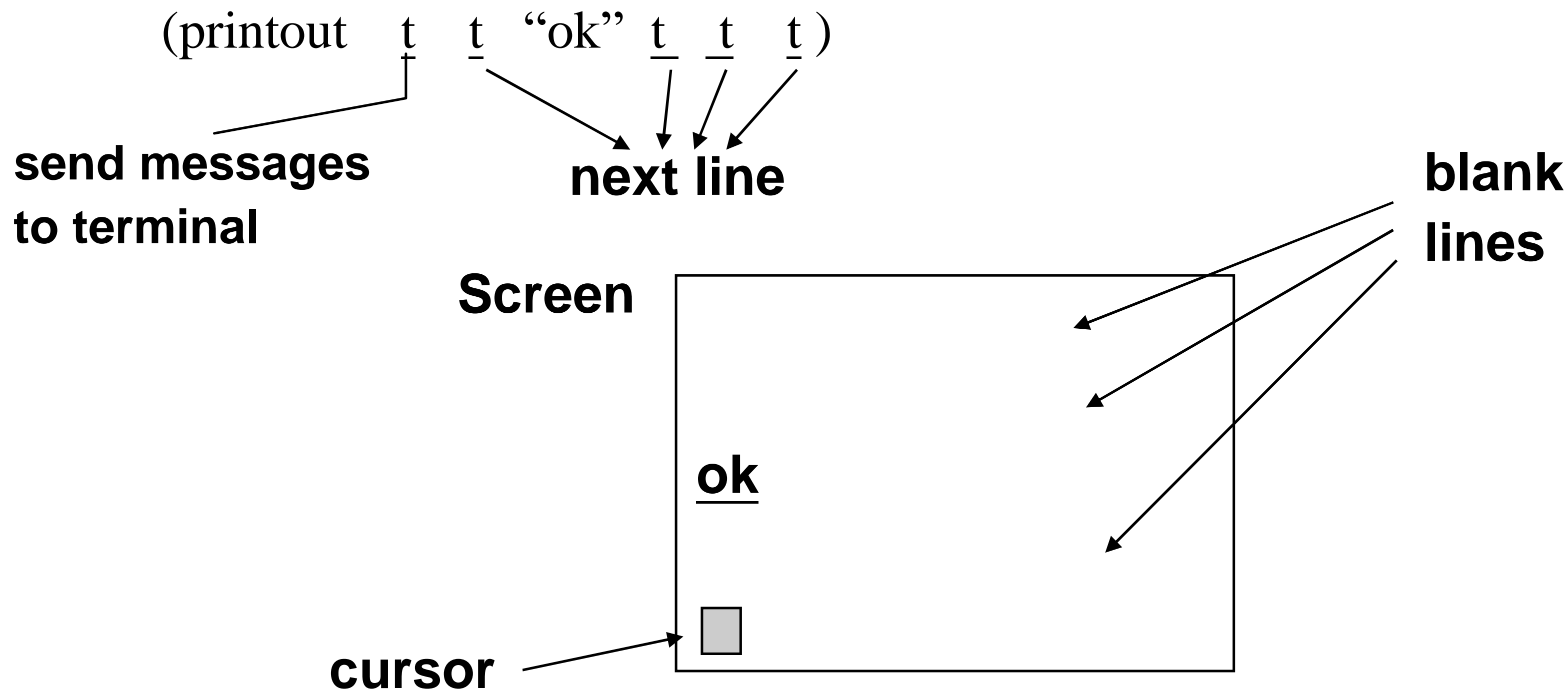
```
(defrule  player-select
     (phase  choose-player)
    =>
     (printout  t "who moves first (c/h) ?")
     (assert (player-select =(READ)))
)
```

*initial phase*

```
(defrule  good-player-choice
     ?phase  <-  (phase choose-player)
     ?choice <-  (player-select ?player&c|h)
    =>
     (retract  ?phase   ?choice)
     (assert  (player-move   ?player))
)
```

**enter next phase: start the game**

```
(defrule  bad-player-choice
      ?phase <-  (phase choose-player)
      ?choice <- (player-select  ?player&~c&~h)
    =>
      (retract  ?phase   ?choice)
      (assert  (phase  choose-player))
      (printout  t  "Stupid! Choose again!" crlf)
)
```



- Predicate (boolean) Functions

  (p.444 ~ p.445)

  NOT
  AND
  OR
  EQ equal (any)
  NEQ not equal (any)
  = equal (numeric only)
  != not equal (numeric only)
  >=
  <=
  <
  >

```
(EQ John John)      ⇨ T
(<   6      8   )   ⇨ T
(=   5      4   )   ⇨ F
(AND  (<   4   5  )
      (=   6   7 )) ⇨ F
(NOT  (!=  6  5))   ⇨ F
```

numberp

stringp

wordp

integerp

evenp

oddp

type predicate functions

(numberp 5 ) ⇨ T

(numberp John) ⇨ F

(stringp "ok") ⇨ T

(evenp 5 ) ⇨ F

```
(defrule  get-human-move
     (player-move  h)

     (pile-size  ?size)
     (test  (> ?size 1))
    =>
     (printout  t   "How many sticks do you  want to take?")
     (assert (human-takes =(read)))
)
```

*matching*

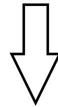
*test condition*

```
(defrule  good-human-move
     ?whose-turn <-  (player-move h)
     ?pile <- (pile-size  ?size)
     ? number-taken  <- (human-take  ?choice)
     (test  (and   (integerp  ?choice)
                   (>=  ? choice 1)
                   (<=  ?choice  3)
                   (<    ?choice  ?size)))
   =>
     (retract  ?whose-turn  ?number-taken ?pile)
     (assert (pile-size =(- ?size ?choice)))
     (assert (player-move c ))
)
```

*test valid moves*

- Predicate field constraint ?

```
(pile-size   ?size)
(test   (>  ?size  1))
```



```
(pile-size  ?size&:(>  ?size  1))
```

*only the fact which satisfies the constraint will be matched*

```
(defrule  add-sum
   (data-item ?value&:(numberp  ?value))
   ?old-total  <- (total  ?total)
  =>
   (retract  ?old-total)
   (assert  (total  =(+  ?total  ?value)))
)
```

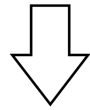
- Equality Field Constraint

"="

```
(defrule computer-move
   ?whose-turn <- (player-move C)
   ?pile <- (pile-size  ?size)
   (test    (>  ?size  1))
   (computer-take  ?number sticks-remained =(mod ?size  4))
  =>
   (retract  ?whose-turn  ?pile)
   (assert (pile-size =(- ?size ?number)))
   (assert  (player-move h))
)
```

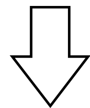
```
(computer-take   1   sticks-remained  1)
(computer-take   1   sticks-remained  2)
(computer-take   2   sticks-remained  3)
(computer-take   3   sticks-remained  0)
```

```
(defrule   test
    (data  length  ?y)
    (data  width    ?x&=(+  9  ?y) | =(- 8 ?y))
   =>
)
```



```
(data  length 4)
```

```
(defrule test
    (data  width  ?x&9|8)
   =>
)
```



```
(data width 9)        matched !
(data width 8)        matched !
```

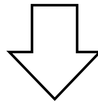
- Salience

```
(defrule fire-first
     (declare (salience 30))
     (priority first)
    =>
     (printout t "print first" crlf)
)
```

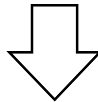
$$-10000 \leq \text{Salience} \leq 10000$$

default = 0

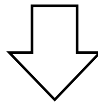
- The main advantage of a rule-based program is that *"programmer does not have to worry about controlling execution"*



*The execution flow is fully controlled by inference engine.*



*The programmer does not need to specify which rule should be fired next.*



*Using too many "Salience" will violate this advantage and make the expert system work like a conventional program.*

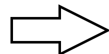
- The uses of salience

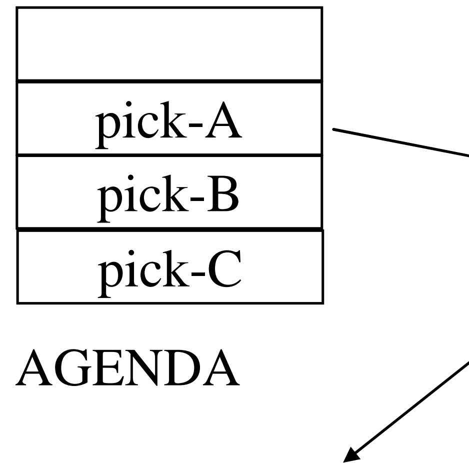
  Salience should primarily be used to determine the order of firing rules.

  Salience should NOT be used as

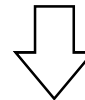
  a method of ***selecting rules to fire***.

```
(defrule pick-A
    (declare (salience 10))
    ?phase <- (choose-move)
    (ready A)
   =>
    (retract  ?phase)
    (assert  (picked A))
)

(defrule  pick-B
    (declare (salience 5))
    ?phase <- (choose-move)
    (ready B)
   =>
    (retract  ?phase)
    (assert  (picked B))
)

(defrule  pick-C
    ?phase <- (choose-move)
    (ready C)
         ...
```

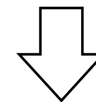





*If rule **pick-A** is fired, rules **pick-B** and **pick-C** will be removed from AGENDA since (choose move) is removed.*



*In this case, Saliences are used to imply **if A, B and C are all true, choose A first***

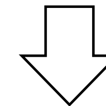


BAD

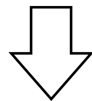
A rule should be independent with any other rule.

```
(defrule pick-B
    (declare (salience 5))
    ?phase <- (choose-move)
    (ready B)
    =>
    (retract ?phase)
    (assert (picked B))
)
```

*IF both A and B are ready, choose A.*

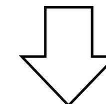


*The meaning of this rule is dependent on rule* ***pick-A***.

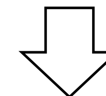


```
(defrule pick-B
     ?phase <- (choose-move)
     (ready B)
     (not (ready A))
    =>
     (retract ?phase)
     (assert (picked B))
 )
```

*GOOD !*

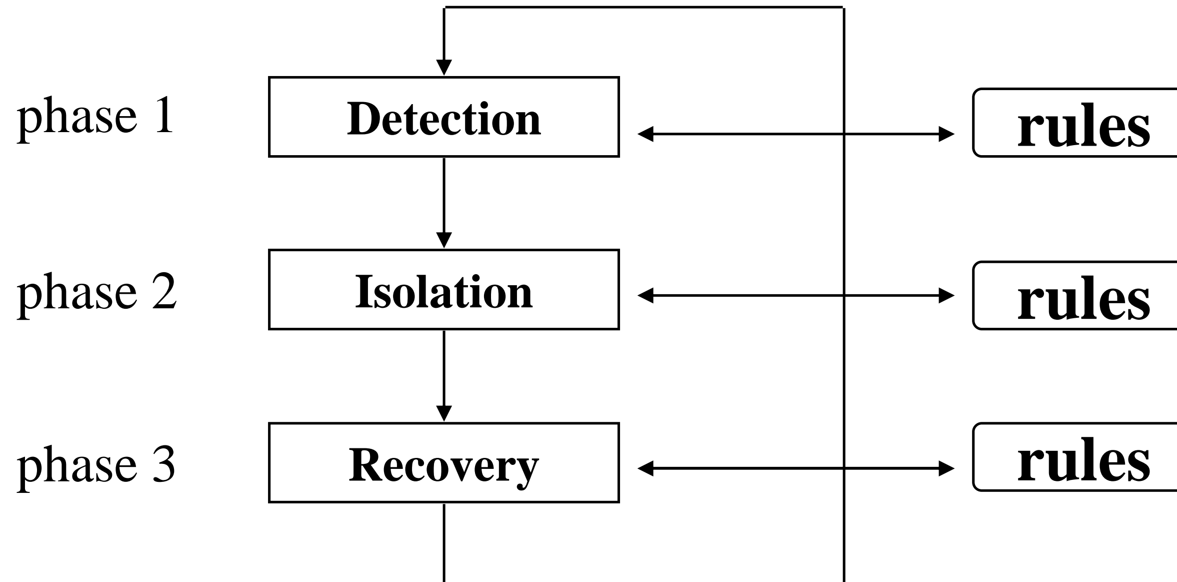


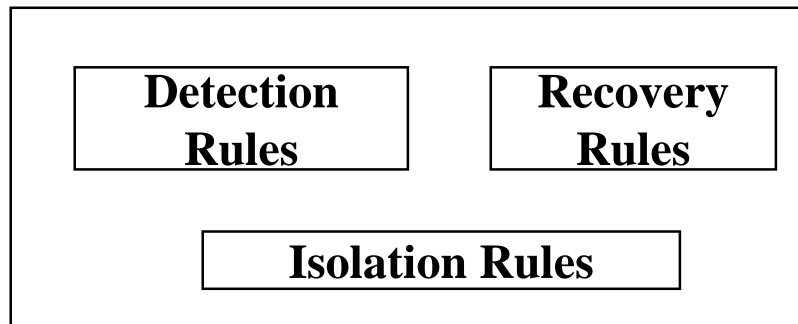
*If* ***pick-A*** *is deleted, or new rules with higher saliences are inserted, the meanings may be changed.*



*Not easy to maintain!*

- Phase and Control Facts



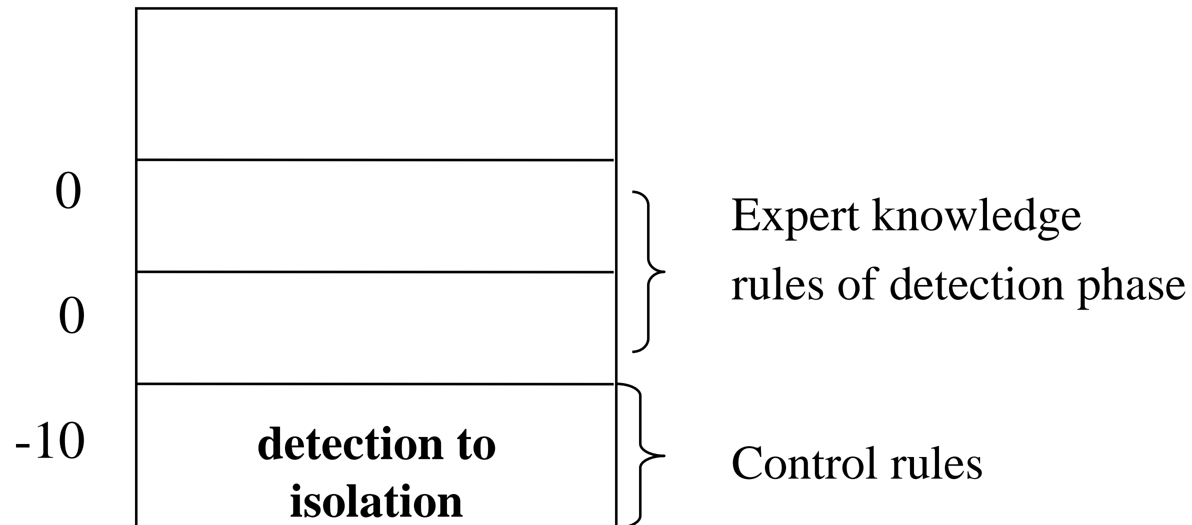

**Expert Knowledge**

**Control Knowledge**

- **Control rules**

```
(defrule detection-to-isolation
   (declare (salience -10))
    ?phase <-   (phase detection)
   =>
    (retract ?phase)
    (assert (phase isolation))
)
```

Fact: ( phase detection)



```
(defrule isolation-to-recovery
    (declare (salience  -10))
     ?phase <-  (phase isolation)
    =>
     (retract ?phase)
     (assert (phase recovery))
)

(defrule recovery-to-detection
        :
        :
)
```

**Control rules**

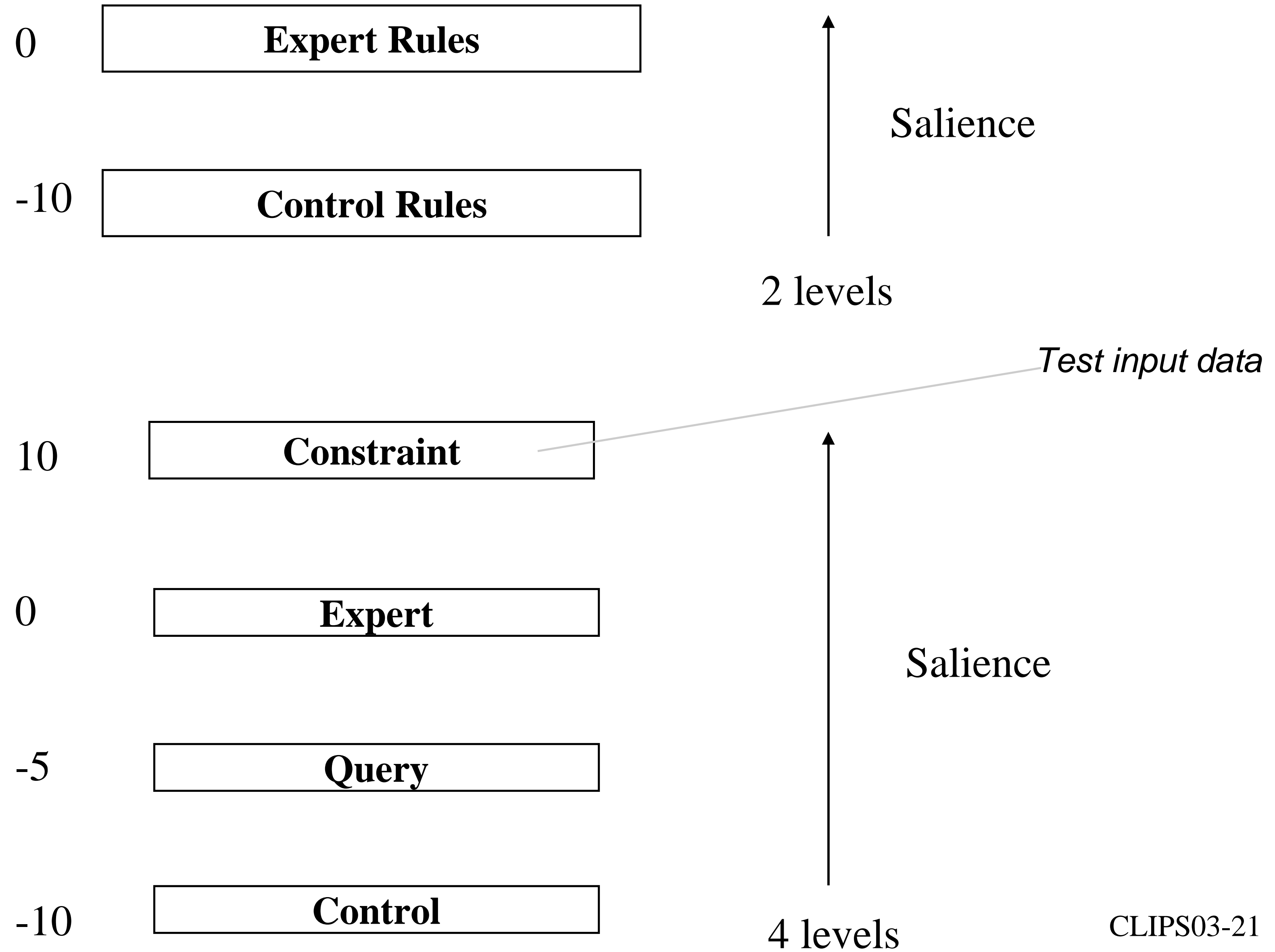
A rule of recovery phase

```
(defrule find-fault-location-and-recovery
    (phase recovery)
    (recovery-solution switch-device ?x on)
    =>
    (printout t "Switch device?" ?x "on" crlf)
)
```

```
CLIPS> (assert (phase detection))
CLIPS> (watch rules)
CLIPS> (run 10)

FIRE   1   detection-to-isolation : f-1
FIRE   2   isolation-to-recovery  : f-2
FIRE   3   recovery-to-detection : f-3
FIRE   4   detection-to-isolation : f- 4
FIRE   5   isolation-to-recovery  : f-5
                    .
                    .

FIRE 10   detection-to isolation : f-10

rule firing limit reached
10 rules fired

CLIPS> ▨
```

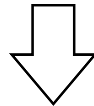
```
(deffacts Control-Information
    (phase detection)
    (phase-after detection isolation)
    (phase-after isolation recovery)
    (phase-after recovery detection)
)

(defrule change-phase
  (declare (salience -10))
  ?phase  <- (phase ?current-phase)
  (phase-after ?current-phase  ?next)
 =>
  (retract ?phase)
  (assert (phase  ?next))
)
```

- Salience hierarchy



- Pattern Logical "OR"

```
(defrule shut-off-electricity
    (or (emergency flood)
        (fire-class C)
        (sprinkler-system active)
    )
   =>
    (printout t "Shut off the electricity" crlf)
)
```



```
        (defrule shut-off-electricity
           (electricity-power on)
AND         (or (emergency flood)
                 (fire-class C)                OR
                 (sprinkler-system active)
            )
          =>
           (printout t "Shut off …" crlf)
        )
```

```
(defrule shut-off
      ?power <- (electricity-power on)
      (or ?reason <- (emergency flood)
            ?reason <- (fire-class C)
           ?reason <- (sprinkler-system active)
      )
     =>
     (retract ?power ?reason)
     (assert (electrical-power off))
)

(defrule shut-off
     ?power  <-  (electrical-power on)
     (or ?reason1 <- (emergency flood)
           ?reason2 <- (fire-class C)
           ?reason3 <- (sprinkler-system active)
     )
    =>
     (retract ?power ?reason1 ?reason2 ?reason3)
     (assert (electrical-power off))
)
```

- Pattern Logical "AND"

```
(defrule electrical-fire
     (emergency fire)
     (fire-class C)
   =>
     (printout t "Shut off the electricity" crlf)
)

            ||

(defrule electrical-fire
    (and (emergency fire))
           (fire-class C)
    )
  =>
    (printout t "Shut off the electricity" crlf)
)
```

```
(defrule shut-off-electricity
    ?power  <- (electrical-power on)
    (or (emergency flood)              ①
        (and (emergency fire)     ⎫
             (fire-class C)       ⎭    ②
        )
        (sprinkler-system active)      ③
    )
  =>
    (retract ?power)
    (assert (electrical-power off))
    (printout t "shut off the electricity" crlf)
)
```

```
(defrule shut-off-1
    ?power <- (electrical-power on)
    (emergency flood)
   =>                                        ①
    (retract ?power)

)
(defrule shut-off-2
   ?power <- (electrical-power on)
   (emergency fire)  ┐
   (fire-class C)    ┘                       ②
  =>
   (retract ?power)
)
(defrule shut-off-3
   ?power  <- (electrical-power on)
   (sprinkler-system active)                 ③
  =>
   (retract ?power)
              :
)
```

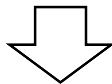
- Pattern Logical "NOT"

```
(defrule no-emergency
     (report-status)
     (not  (emergency ?))
   =>
     (printout t "No emergency" crlf)
)
```

*No such fact exists*

```
(defrule Largest-number
     (number ?x&:(numberp ?x))
     (not (number  ?y&:(> ?y ?x)))
   =>
     (printout t "Largest number is" ?x  crlf)
)
```

*There does not exist some ?y, such that ?y > ?x.*



*?x should be the maximum value*

EX. 9-5 Is the variable X referenced properly for the following rules?

1.
```
(defrule example-1
        (not (fact ?x))
        (test (> ?x 4))
   =>
              :
 )
```
2.
```
(defrule example-2
        (not (fact ?x &:(> ?x 4)))
   =>
              :
 )
```
3.
```
(defrule example-3
        (not (fact ?x))
        (fact ?y &: (> ?y ?x))
   =>
              :
 )
```
4.
```
(defrule example-4
        (not (fact ?x))
        (fact ?x &: (> ?x 4))
   =>
              :
 )
```