CLIPS Tutorial 4

Modular Design, Execution Control, and Rule Efficiency

Deftemplate Attributes

- CLIPS provides a number of slot attributes that can be specified when a deftemplate's slots are defined.
- It is possible to define the allowed types and values that can be stored in a slot.

The Type Attribute

- The type attribute defines the data types that can be placed in a slot.
- The general format of the type attribute is

  `(type <type-specification)`

  - where <type-specification> is either
    - ?VARIABLE or
    - one or more of the symbols
      - SYMBOL,
      - STRING,
      - LEXEME,
      - INTEGER,
      - FLOAT,
      - NUMBER,
      - INSTANCE-NAME,
      - INSTANCE-ADDRESS,
      - INSTANCE,
      - FACT-ADDRESS, or
      - EXTERNAL-ADDRESS.

The Type Attribute

- Example:

```
(deftemplate person
    (multislot name (type SYMBOL))
    (slot  age (type INTEGER)))
```

- For example, assigning the symbol four to the age slot rather than the integer 4 will cause an error as shown:

```
CLIPS> (assert (person (name Fred Smith)
                       (age four)))
```

[CSTRNCHK1] A literal slot value found in the assert command does not match the allowed types for slot age.

CLIPS>

The Type Attribute

- CLIPS also checks the consistency of bindings in the LHS and RHS of a rule
- Example:

```
(deftemplate had-a-birthday
    (slot name (type STRING)))
```

- The following will cause an error:

```
CLIPS> (defrule update-birthday
         ?f1 <- (had-a-birthday (name ?name))
         ?f2 <- (person (name ?name) (age ?age))
         =>
         (retract ?f1)
         (modify ?f2 (age (+ ?age 1))))
```

[RULECSTR1] Variable ?name in CE #2 slot name has constraint conflicts which make the pattern unmatchable.

CLIPS>

The Allowed Value Attributes

- CLIPS also allows you to specify a list of allowed values for a specific type.
- For example, if a *gender* slot is added to the *person* deftemplate, the allowed symbols for that slot can be restricted to *male and female:*

```
(deftemplate person(
  multislot name (type SYMBOL))
  (slot age (type INTEGER))
  (slot gender (type SYMBOL)
               (allowed-symbols male female)))
```

The Allowed Value Attributes

- There are eight different allowed value attributes provided by CLIPS:
  - allowed-symbols, allowed-strings, allowed-Lexemes, allowed-integers, allowed-floats, allowed-numbers, allowed-instance-names, and allowed-values.
- For example, (allowed-symbols male female) does not restrict the type of the *gender* slot to being a symbol.
  - It merely indicates that if the slot's value is a symbol, then it must be one of the two symbols: either *male* or *female*.
  - Any string, integer, or float would be a legal value for the *gender* slot if the (type SYMBOL) attribute were removed
- The **allowed-values** attribute can be used to completely restrict the set of allowed values for a slot to a specified list.

```
(deftemplate person
        (multislot name (type SYMBOL))
        (slot age (type INTEGER))
        (slot gender (allowed-values male female)))
```

The Range Attribute

- The range attribute allows the specification of minimum and maximum numeric values.
- The general format of the range attribute is

  (range <lower-limit> <upper-limit>)
  - where <lower-limit> and <upper-limit> are either ?VARIABLE or a numeric value.
  - ?VARIABLE indicates there is either no maximum or minimum value.
- For example, to prevent negative values:

```
(deftemplate person
    (multislot name (type SYMBOL))
    (slot age (type INTEGER) (range 0 ?VARIABLE)))
```

The Cardinality Attribute

- The cardinality attribute allows the specification of the minimum and maximum number of values that can be stored in a multislot.
- The general format of the cardinality attribute is
  `(cardinality <lower-limit> <upper-limit>)`
  - where<lower-limit> and <upper-limit> are either ?VARIABLE or a positive integer.
  - ?VARIABLE indicates there is either no maximum or minimum value.
- Note that type, allowed value, and range attributes are applied to every value contained in a multislot.
- Example:

```
(deftemplate volleyball-team
    (slot name (type STRING))
    (multislot players (type STRING)
                       (cardinality 6 6))
    (multislot alternates (type STRING)
                       (cardinality 0 2)))
```

The Default Attribute

- It is often convenient to automatically have a specified value stored in a slot if no value is explicitly stated in an *assert* command.
- The general format of the default attribute is

  (default <default-specification>)
  - where<default-specification> is either ?DERIVE, ?NONE, a single expression (for a single-field slot), or zero or more expressions (for a multifield slot).
- If the default attribute is not specified for a slot, then it is assumed to be (default ?DERIVE).
  - For a single-field slot, this means that a value is selected that satisfies the type, range, and allowed values attributes for the slot.
  - The derived default value for a multifield slot will be a list of identical values that are the minimum allowed cardinality for the slot (zero by default).

The Default Attribute

- An example of derived values is the following:

```
CLIPS> (clear)
CLIPS>    (deftemplate example
          (slot a)
          (slot b (type INTEGER))
          (slot c (allowed-values red green blue))
          (multislot d)
          (multislot e (cardinality 2 2)
          (type FLOAT)
          (range 3.5 l0.0)))
CLIPS> (assert (example))
<Fact - 0>
CLIPS> (facts)
f – 0     (example  (a nil)
                    (b 0)
                    (c red)
                    (d)
                    (e  3.5 3.5))
For a total of 1 fact.
CLIPS>
```

The Default Attribute

- If ?NONE is specified in the default attribute. a value must be supplied for the slot when the fact is asserted.

```
CLIPS> (clear)
CLIPS>
(deftemplate example
(slot a)
(slot b (default ?NONE)))
CLIPS> (assert (example))
[TMPLTRHS1] Slot b requires a value because of its
(default ?NONE) attribute.
CLIPS> (assert (example (b 1)))
<Fact-0>
CLIPS> (facts)
f-0(example (a nil) (b 1))
For a total of 1 fact.
CLIPS>
```

The Default Attribute

- An example using expressions with the *default* attribute:

```
CLIPS> (clear)
CLIPS>
(deftemplate example
   (slot a (default 3))
   (slot b (default (+ 3 4)))
   (multi slot c (default a b c))
   (multi slot d (default (+ 1 2) (+ 3 4))))
CLIPS> (assert (example))
<Fact-0>
CLIPS> (facts)
f-0       (example (a 3) (b 7) (c a b c) (d 3 7))
For a total of 1 fact.
CLIPS>
```

The Deffunction Construct

- New functions are defined using the deffunction construct.
- The general format of a deffunction is:

```
(deffunction <deffunction-name> [<optional-comment>]
     (regular-parameter>* [<wildcard-parameter>])
     <expression>*)
```

  - Where <regular-parameter> is a single-field variable and <wildcard-parameter> is a multifield variable.
  - The name of the deffunction, <deffunction-name>, must be distinct.
  - The body of the deffunction, represented by <expression>*, is a series of expressions similar to the RHS of a rule that are executed in order when the deffunction is called.
  - Unlike predefined functions, deffunctions can be delted and the watch command can be used to trace their execution.

The Deffunction Construct

- The <regular-parameter> and <wildcard-parameter> declarations
- Specify the arguments that will be passed into the deffunction when it is called.

- A deffunction can return values.
  - The return value is that value of the last expression evaluated within the body of the deffunction.

- E.g.

```
(deffunction hypotenuse-length (?a ?b)
    (** (+ (* ?a ?a) (* ?b ?b)) 0.5))
```

  - Where the **function with its second argument of 0.5
    - compute the square root
    - (** <numeric-expression> <numeric-expression>) is the first argument raised to the power of the second argument

The Deffunction Construct

- It can be called from the command prompt:

```
CLIPS> (hypotenuse-length 3 4)
5.0
CLIPS>
```

- In a more readable format:

```
(deffunction hypotenuse-length (?a ?b)
   (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
   (** ?temp 0.5))
```

The Deffunction Construct

- The *Return* Function
  - It allows the currently executing deffunction to be terminated.
  - Its syntax for use with deffunctions:

    ```
    (return [<expression>])
    ```

  - If <expression> is specified, the result of its evaluation is used as the return value.
  - E.g.

```
(deffunction hypotenuse-length (?a ?b)
     (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
     (return (** ?temp 0.5)))
OR
(deffunction hypotenuse-length (?a ?b)
     (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
     (bind ?c (** ?temp 0.5))
     (return ?c))
```

The Deffunction Construct

- Watching Deffunctions
  - When deffunctions are watched using watch command, an informational message is printed whenever a deffuction begins or ends execution.

```
CLIPS> (watch deffucntions)
CLIPS> (unwatch deffunctions)
```

  - Watch specific deffucntions

```
CLIPS> (watch deffucntions hypotenuse-length)
CLIPS>
```

- Wildcard parameter
  - If the last parameter declared in a deffunction is a multifield variable, which is referred to as a wildcard parameter, then the deffunction can be called with more arguments than are specified in the parameter list.

The Deffunction Construct

- Deffunction commands
  - Display the text representations of a deffunction:

    (ppdeffunction <deffunction-name>)
  - Delete a deffunction:

    (undeffunction <deffunction-name>)
  - Display the list of deffunctions defined:

    (list-deffunctions [<module-name>])
  - Returns a multifield value containning the list of deffucntions.

    (get-deffunction-list [<module-name>])

The Defglobal Construct

- Global variables:
  - CLIPS allows one to define variables that retain their values outside the scope of a construct
  - Local variables:

```
(defrule example-1
   (data-1 ?x)
   =>
   (printout t "?x = " ?x crlf))
(defrule example-1
   (data-2 ?x)
   =>
   (printout t "?x = " ?x crlf))
```

    - The value of ?x in rule example-1 does not constrain in any way the value of ?x in rule example-2.

The Defglobal Construct

- The general format of a defglobal is:

  (defglobal [<defmodule-name>] <global-assignment>*)

  - Where <global-assignment> is:

    <global-variable> = <expression>

  - And <global-variable> is:

    ?*<symbol>*

  - Global variable names begin and end with the * character.
    - ?x is a local variable
    - ?*x* is a global variable.
    - E.g.

```
CLIPS> (defglobal ?*x* = 3
                  ?*y* = (+ ?*x* 1))
CLIPS> ?*x*
3
CLIPS> ?*y*
4
CLIPS>
```

The Defglobal Construct

- Example:

```
CLIPS> (defrule area
(radius ?r)
=>
(bind ?area (* ?*pi* ?*pi* ?r))
(printout t "Area = " ?area crlf))
CLIPS> (deffacts area_circle (radius 4))
CLIPS> (reset)
CLIPS> (run)
Area = 39.47841751413609
CLIPS>
```

- The value of a defglobal can be changed using the bind command.

The Defglobal Construct

- Defglobals can be used in expression on the LHS of rules, but changes to defglobals won't trigger pattern matching.
- Example:

```
(defglobal ?*z* = 4)
(defrule global-example
   (data ?z&:(> ?z ?*z*))
   =>)
```

The Defglobal Construct

- When facts are asserted:

```
CLIPS> (reset)
CLIPS> ?*z*
4
CLIPS> (assert (data 5) (data 6))
<Fact-2>
CLIPS> (facts)
f-0       (initial-fact)
f-1       (data 5)
f-2       (data 6)
For a total of 3 facts.
CLIPS> (agenda)
0         global-example: f-1
0         global-example: f-2
For a total of 2 activations.
CLIPS> (bind ?*z* 5)
5
CLIPS> (agenda)
0         global-example: f-1
0         global-example: f-2
For a total of 2 activations.
```

- As pattern matching is complete when facts are asserted, changing the value of ?*Z* won't cause the pattern to be reevaluated.

The Defglobal Construct

- Defglobals are most appropriately used in rules as either
  - constants or
  - to pass in information that is used only on the RHS of the rule

  and should not trigger pattern matching.

Salience

- CLIPS provides two explicit techniques for controlling the execution of rules: salience and modules.
- The use of the keyword *salience* allows the priority of rules to be explicitly specified.
- Normally the agenda acts like a stack.
  - the most recent activation placed on the agenda is the first to fire.
- Salience allows more important rules to stay at the top of the agenda, regardless of when the rules were added.
- Salience is set using a numeric value ranging from the smallest value of -10,000 to the highest of 10,000.
  - If a rule has no salience explicitly assigned by the programmer, CLIPS assumes a salience of 0.
- A newly activated rule is placed on the agenda before all rules with equal or lesser salience and after all rules with greater salience.

Salience

- No salience values are declared:

```
(defrule fire-first
    (priority first)
    =>
    (printout t "Print first" crlf))
(defrule fire-second
    (priority second)
    =>
    (printout t "Print second" crlf))
 (defrule fire-third
    (priority third)
    =>
    (printout t "Print third" crlf))
```

Salience

Produce the output shown below:

```
CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (assert (priority first))
<Fact-1>
CLIPS> (assert (priority second))
<Fact-2>
CLIPS> (assert (priority third))
<Fact-3>
CLIPS> (run)
Print third
Print second
Print first
CLIPS>
```

Salience

- By declaring salience values:

```
(defrule fire-first
  (declare (salience 30))
  (priority first)
  =>
  (printout t "Print first" crlf))
(defrule fire-second
  (declare (salience 20))
  (priority second)
  =>
  (printout t "Print second" crlf))
(defrule fire-third
  (declare (salience 10))
  (priority third)
  =>
(printout t "Print third" crlf))
```

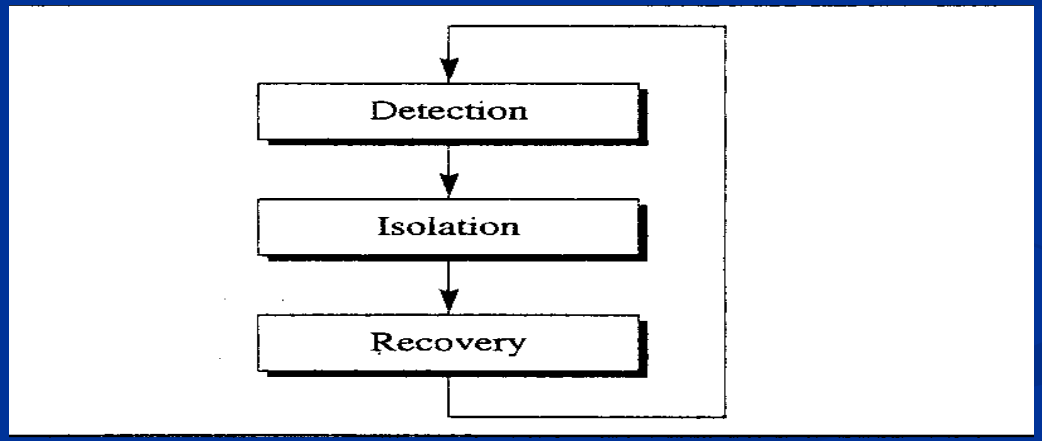
Salience

- Produce the following output:

```
CLIPS> (reset)
CLIPS> (assert (priority second)
               (priority first)
               (priority third))
<Fact-3>
CLIPS> (agenda)
30 fire-first:     f-2
20 fire-second:    f-1
10 fire-third:     f-3
For a total of 3 activations.
CLIPS>
```

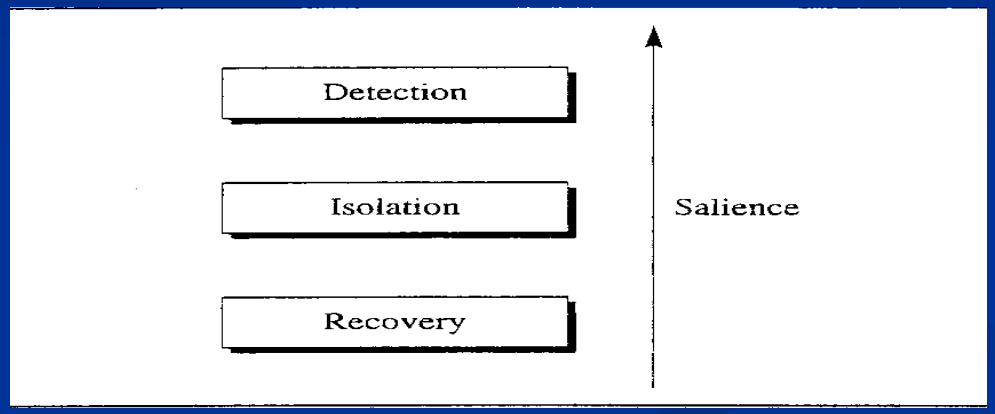
Phases and Control Facts

- For programs involving hundreds or thousands of rules, the intermixing of domain knowledge and control knowledge makes development and maintenance a major problem.
- As an example, consider the problem of performing fault detection, isolation, and recovery of a system such as an electronic device.
  - Different Phases for Fault Detection, Isolation, and Recovery Problem



Phases and Control Facts

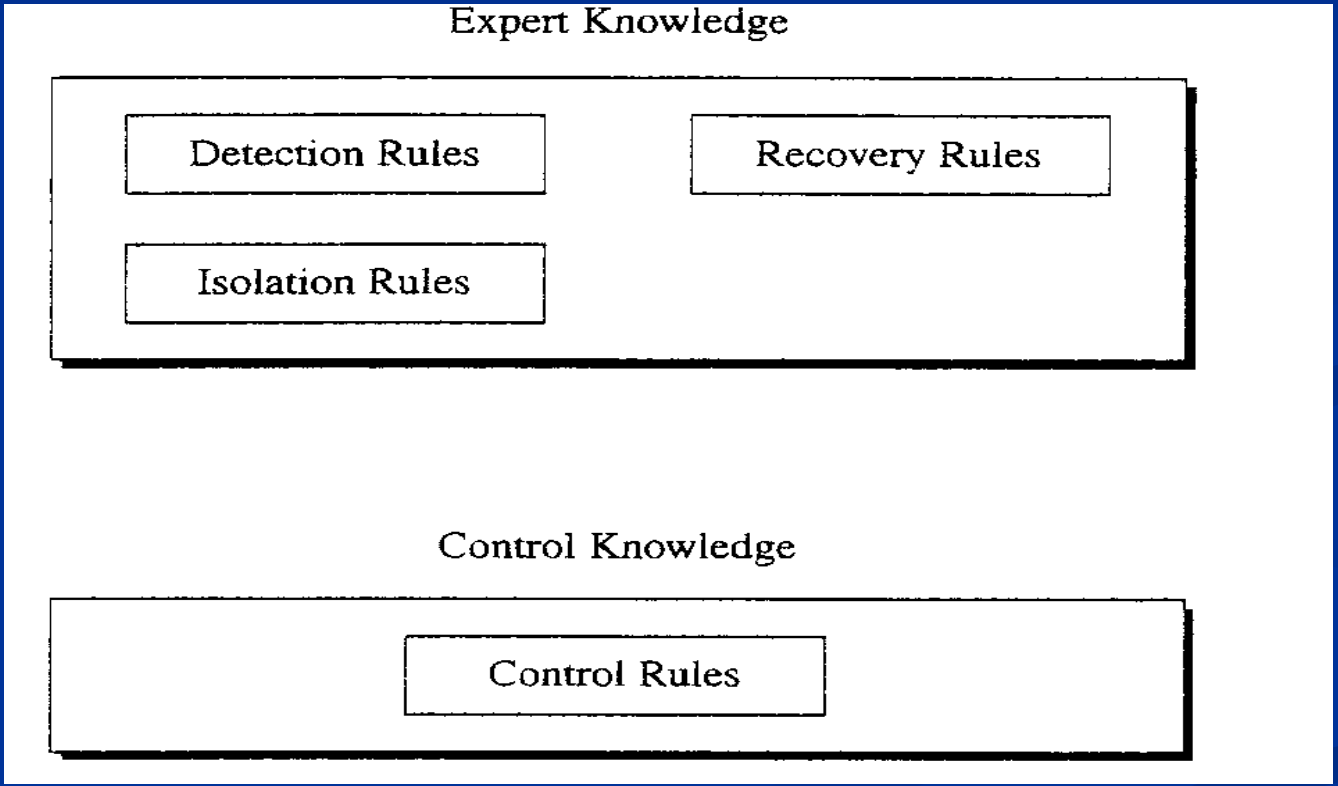
- To use salience to organize the rules,
  - Assignment of salience for different phases



  - two major drawbacks are:
    - control knowledge is still being embedded into the rules using salience.
    - does not guarantee the correct order of execution.
- Better approach in controlling the flow of execution is to separate the control knowledge from the domain knowledge, as shown in the following figure.
  - each rule is given a control pattern that indicates its applicable phase.
  - Control rules are then written to transfer control between the different phases

Phases and Control Facts

Separation of Expert Knowledge from Control Knowledge



Phases and Control Facts

- Control rules:

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

Phases and Control Facts

- Each of the rules applicable for a particular phase is then given a control pattern.

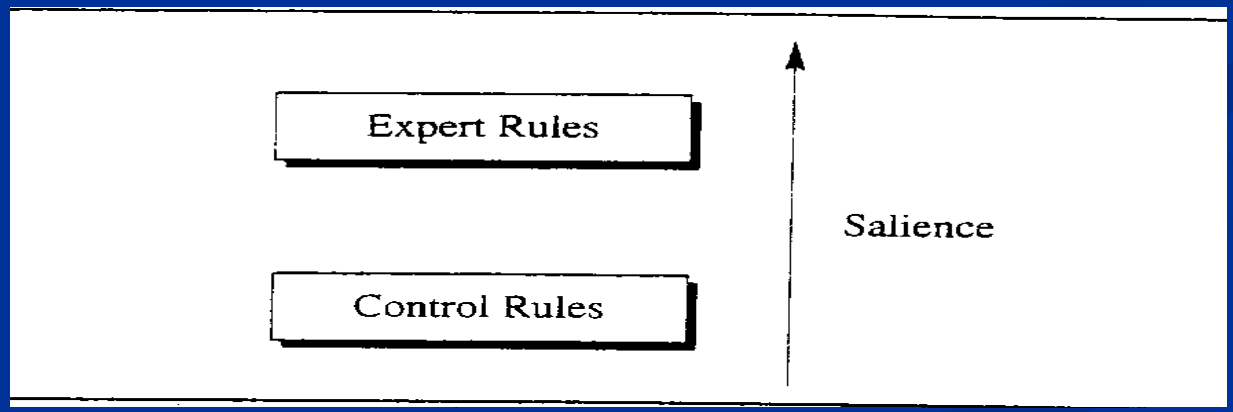
```
(defrule find-fault-location-and-recovery
        (phase recovery)
        (recovery-solution switch-device
                ?replacement on)
        =>
        (printout t "Switch device" ?replacement "on"
                        crlf) )
```

- A salience hierarchy is a description of the salience values used by an expert system.
  - Each level in a salience hierarchy corresponds to a specific set of rules whose members are all given the same salience.

Phases and Control Facts

- While the fact (phase detection) is in the fact list, the *detection-to-isolation* rule will be on the agenda.
  - Since it has a lower salience than the detection rules, it will not fire until all of the detection rules have had an opportunity to fire.

Salience Hierarchy Using Expert and Control Rules



Phases and Control Facts

- The previous rules could be more generally written as:

```
(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-phase)
        =>
        (retract ?phase)
        (assert (phase ?next-phase))
```

Phases and Control Facts

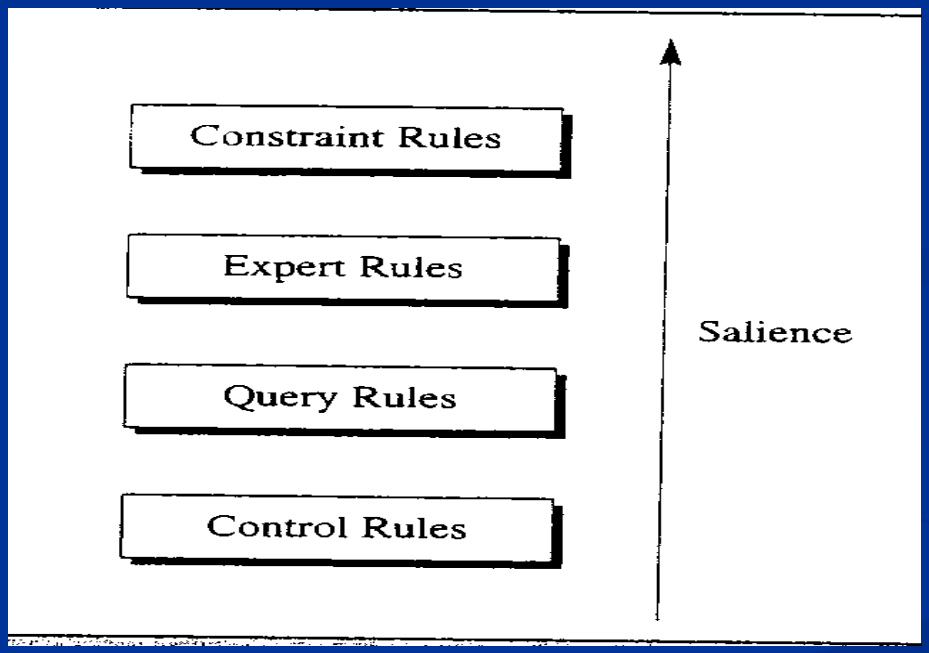
- Or as a sequence of phases:

```
(deffacts control-information
        (phase detection)
        (phase-sequence isolation recovery detection))
(defrule change-phase
        (declare (salience –10))
        ?phase <- (phase ?current-phase)
        ?list <- (phase-sequence ?next-phase ?other-phases)
        =>
        (retract ?phase ?list)
        (assert (phase ?next-phase))
        (assert (phase-sequence ?other-phases ?next-phase)))
```

Phases and Control Facts

- Additional levels can be easily added to the salience hierarchy.

Four-Level Salience Hierarchy



Misuse of Salience

- Overuse of salience results in a poorly coded program.
  - A main advantage of a rule-based program is that the programmer does not have to worry about controlling execution.
- Salience should primarily be used as a mechanism for determining the order in which rules fire.
  - Salience should not be used as a method for selecting a single rule from a group of rules when patterns can be used to express the criteria for selection