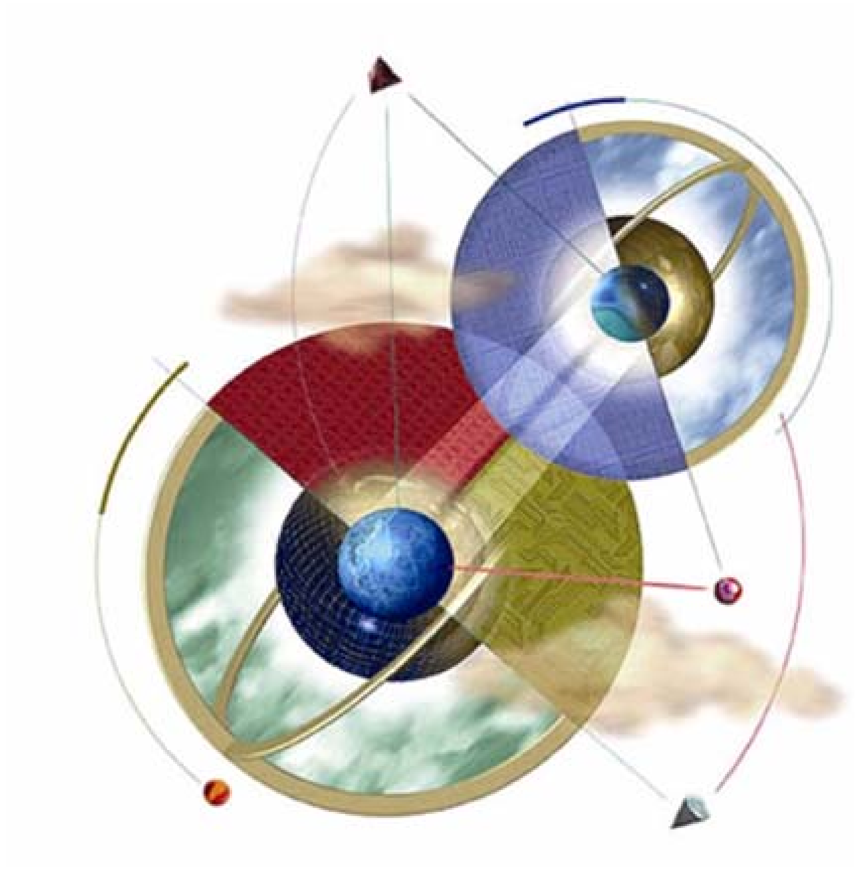


Chapter 9: Modular Design, Execution Control, and Rule Efficiency

Expert Systems: Principles and Programming, Fourth Edition

Deftemplate Attributes

- CLIPS provides slot attributes which can be specified when deftemplate slots are defined.
- Slot attributes provide strong typing and constraint checking.
- One can define the allowed types that can be stored in a slot, range of numeric values.
- Multislots can specify min / max numbers of fields they can contain.
- Default attributes can be provided for slots not specified in an assert command.

Type Attribute

- Defines the data types can be placed in a slot
- Example:

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

- Once defined, CLIPS will enforce these restrictions on the slot attributes

  name – must store symbols

  age – must store integers
- Try to assert a fact
  - (assert (person (name Fred Smith) (age four)))
    - What happened?

Type Attribute cont.

- CLIPS will also check the consistency of ***variable bindings***

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

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

- Will the rule be successfully added into CLIPS?

Static and Dynamic Constraint Checking

- CLIPS provides two levels of constraint checking (type checking)
  - Static constraint checking
    - Performed when CLIPS parses expression (compile time)
    - Can be enabled/disabled by calling the *set-static-constraint-checking* function and passing it TRUE/FALSE
      - enabled by default
      - (set-static-constraint-checking FALSE)
      - (get-static-constraint-checking) for checking the status
  - Dynamic constraint checking
    - Performed on facts when they are asserted (run time)
    - Can be enabled / disabled with *set-dynamic-constraint-checking*
      - disabled by default
      - (set-dynamic-constraint-checking TRUE)
      - (get-dynamic-constraint-checking) for checking the status

Why Dynamic Constraint Checking

- (defrule create-person
-   =>
-   (printout t “what is your name? “)
-   (bind ?name (explode$ (readline)))
-   (printout t “What is your age? “)
-   (bind ?age (read))
-   (assert (person (name ?name) (age ?age))))

- CLIPS>(reset)
- CLIPS>(run)
- What is your name? **Fred Smith**
- What is your age? **Four**
- CLIPS>(facts)
- f-0 (initial-fact)
- f-1 (person (name Fred Smith) (age ***four***))
- ……

Violating the constraint that the age slot must be an Integer

Allowed Value Attributes

- CLIPS allows one to specify a list of allowed values for a specific type
- (deftemplate person

  (multislot name (type SYMBOL))

  (slot age (type INTEGER))

  (slot gender (type SYMBOL) (allowed-symbols male female)))
  - allowed-symbols does not restrict the type of the gender slot to being a symbol
    - if the slot’s value is a symbol, then it must be either male or female
    - any string, integer, or float would be a legal value if the (type SYMBOL) were removed
    - (allowed-values male female) can be used to completely restrict the slot, and (type SYMBOL) can be removed in this case

Allowed Value Attributes (cont.)

- CLIPS provides several different allowed value attributes
  - allowed-symbols
  - allowed-strings
  - allowed-lexemes (equals to symbols & strings)
  - allowed-integers
  - allowed-floats
  - allowed-numbers
  - allowed-instance-names
  - allowed-values

Range Attributes

- This attribute allows the specification of minimum and maximum numeric values.
  - (range <lower-limit> <upper-limit>)
- Example:

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER) (range  0  120)))
```

Can be replaced by ?VARIABLE
If not upper-limit is specified

Cardinality Attributes

- This attribute allows the specification of minimum and maximum number of values that can be stored in a multislot.
  - (cardinality <lower-limit> <upper-limit>)
- Example:

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

Default Attribute

- Previously, each deftemplate fact asserted had an explicit value stated for every slot.
- It is often convenient to automatically have a specified value stored in a slot if no value is explicitly stated in an *assert* command.
  - (default <default-specification>)
    - <default-specification> is either **?DERIVE**, **?NONE**, a signle expression or zero or more expressions
  - Example:
    - (deftemplate example (slot aa (default 3)) (slot bb (default 5))
      (multislot cc (default red green blue)))
    - (assert (example (bb 10)))
      - (example (aa 3) (bb 10) (cc red green blue))

Example

- (deftemplate example
  - (slot a)
  - (slot b (type INTEGER))
  - (slot c (allowed-value red green blue))
  - (multislot d)
  - (multislot e (cardinality 2 2)
  - (type FLOAT)
  - (range 3.5 10.0)))
- (assert (example))
- How does the fact look like?
  - If the default attributes is not specified for a slot, then it is assumed to be (default ?DERIVE)

Example

- (deftemplate example
  - (slot a)
  - (slot b (default **?NONE**)))
- (assert (example))
- What happened?

Example

- The default attribute can also be an expression
- (deftemplate example
  - (slot a (default 3))
  - (slot b (defaut (+ 3 4)))
  - (multislot c (default a b c))
  - (multislot d (default (+ 1 2) (+ 3 4))))
- (assert (example))
- (facts)
- (example (a 3) (b 7) (c a b c) (d 3 7)

Default-Dynamic Attribute

default attribute can be dynamic, i.e. time-dependent

- (time) is used to record the time of fact creation
- It returns the number of seconds that have elapsed since the CLIPS system executed
- (deftemplate data (slot value) (slot creation-time (default-dynamic (time))))
- (defrule retract-data-facts-after-one-minute
```
    ?f <- (data (value ?) (creation-time ?t1))
    ?c <- (current-time ?t2)
    (test (> (- ?t2 ?t1) 60))
  =>
    (retract ?f ?c))
```
- (defrule check-current-time
```
    ?c <- (current-time ?t)
  =>
    (retract ?c)
    (assert (current-time (time))))
```

Default-Dynamic Attribute

– What happened if we do the following change to the rule?

– (deftemplate data (slot value) (slot creation-time (default-dynamic (time))))

– (defrule retract-data-facts-after-one-minute

```
        ?f <- (data (creation-time ?t1))
        (test (> (- (time) ?t1) 60))
      =>
        (retract ?f))
```

Conflicting Slot Attributes

CLIPS does not allow you to specify conflicting attributes for a slot.

Salience

- CLIPS provides two explicit techniques for controlling the execution of rules:
  - Salience
  - Modules
- Salience allows the priority of rules to be explicitly specified.
- The agenda acts like a stack (LIFO) – most recent activation placed on the agenda being first to fire.

Salience (cont.)

- Salience allows more important rules to stay at the top of the agenda, regardless of when they were added.
- Lower salience rules are pushed lower on the agenda; higher salience rules are higher.
- Salience is set using numeric values in the range -10,000 → +10,000
- Zero is the default priority.
- Salience can be used to force rules to fire in a sequential fashion.

Salience (cont.)

- Rules of equal salience, activated by different patterns are prioritized based on the stack order of facts.
- If 2+ rules with same salience are activated by the same fact, no guarantee about the order in which they will be place on the agenda.

Salience

- (declare (salience 100))
  - the priority to fire rules (-10000 ... 10000)
  - default is 0
  - (defrule rule-01

```
      (declare (salience 50))
      (person (name ?name) (age 20))
      =>
     (printout t ?name “ is young.” crlf))
(defrule rule-02
      (declare (salience 20))
      (person (name ?name) (age 20))
      =>
     (printout t ?name “ is 20 years old.” crlf))
```

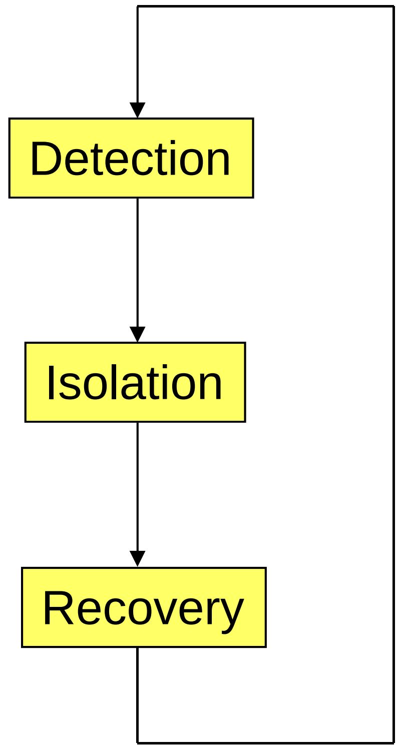
```
f-5 (person (name Peter) (age 20))
f-6 (person (name David) (age 20))
```

Phases and Control Facts

- The purest concept of expert system: the rules act opportunistically whenever they are applicable
- Most expert systems have some procedural aspect to them
  - game of sticks: control facts (player-get c) to control human's move or computer's move
  - a major problem in development and maintenance: control knowledge is intermixed with domain knowledge
  - domain knowledge and control knowledge should be separated

An Example: Electronic Device Problem

- Different phases for solving electronic device problem
  - fault detection
    - recognize that the electronic device is not working properly
  - isolation
    - determine the components of the device that have caused the fault
  - recovery
    - determine the steps necessary to correct the fault



Implementing the Flow of Control in the Electronic Device System

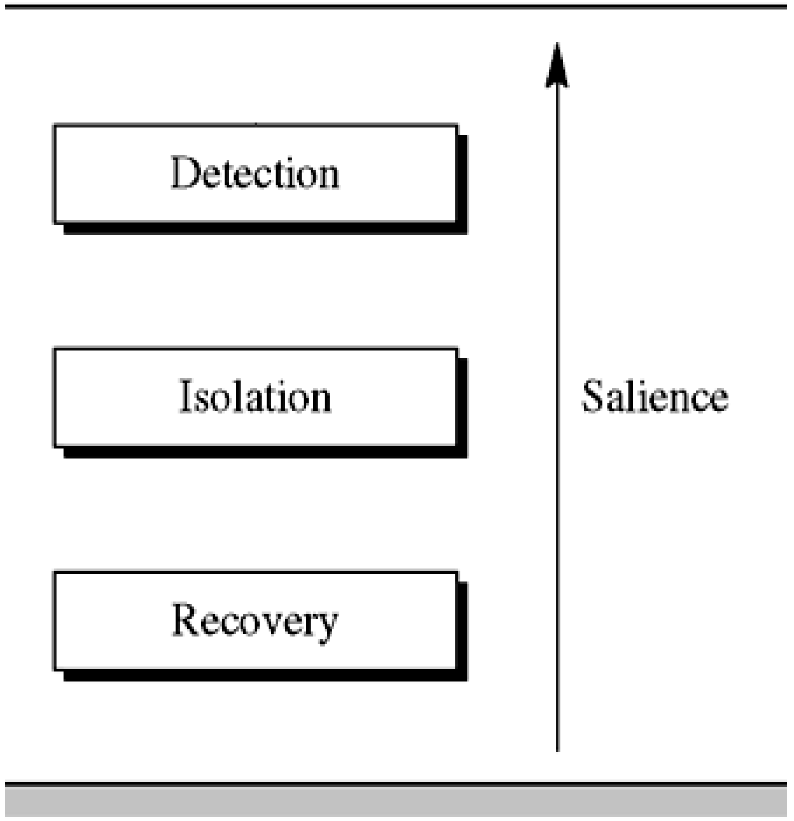
- Four ways to implement the flow of control:
  1. Embed the control knowledge directly into the rules.
  2. Use salience to organize the rules
  3. Separate the control knowledge from the domain knowledge
  4. Use modules to organize the rules (will be discussed latter)

1. Embed the control knowledge directly into the rules

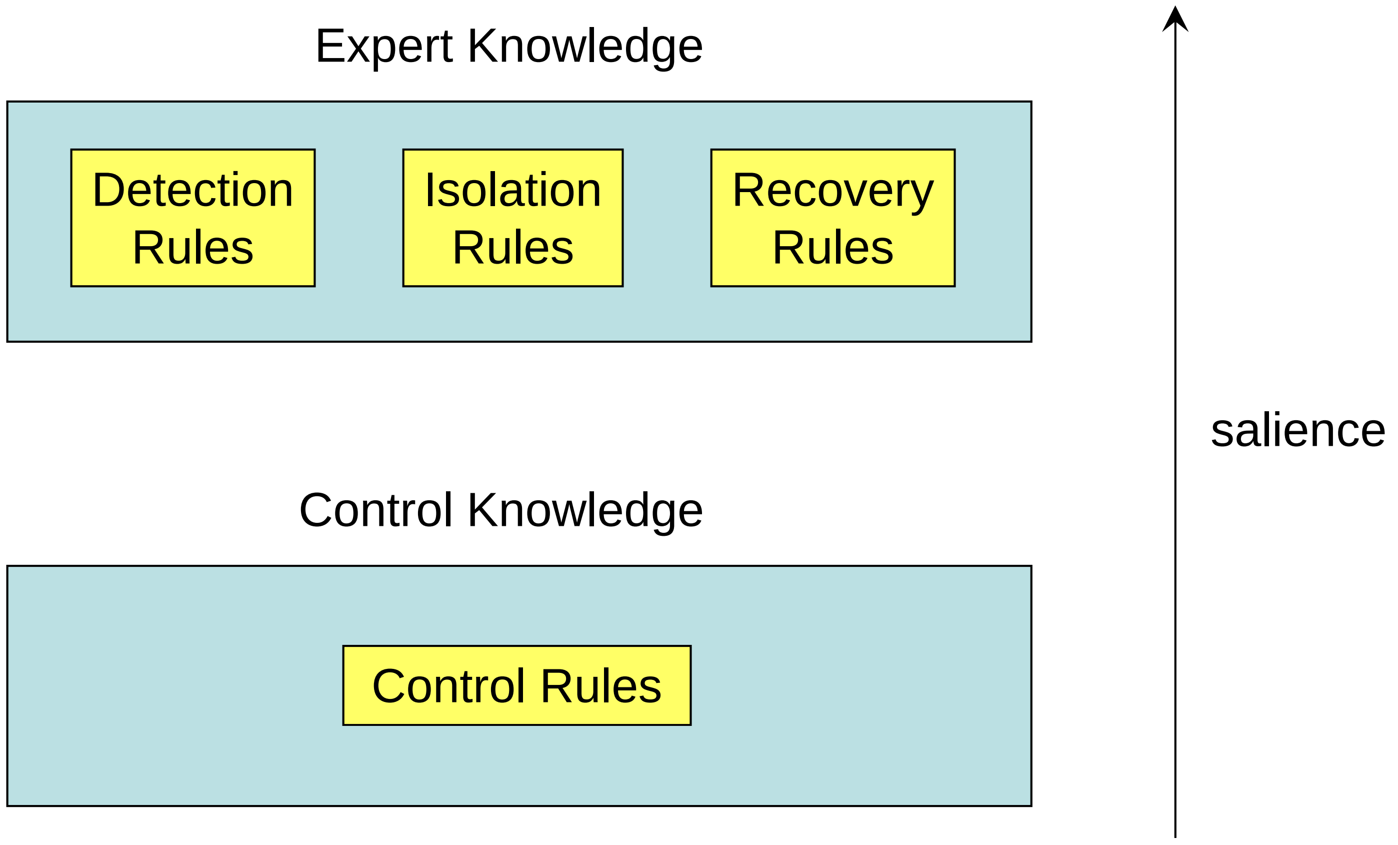
- Each rule would be given a pattern (phase xxx) indicating in which phase it would be applicable.
- Detection rules would include rules indicating when the isolation phase should be entered.
  - retract (phase detection) and assert (phase isolation)
- Drawbacks:
  - intermixing control knowledge and domain knowledge makes it difficult to develop and maintain
  - it is not always easy to determine when a phase is completed

2. Use salience to organize the rules

- Drawbacks:
  - control knowledge is still being embedded into the rules using salience
  - does not guarantee the correct order of execution
    - detection rules always fire before isolation rules
    - the firing of some isolation rules might cause the activation of some detection rules



3. Separate the control knowledge from the domain knowledge



3. Separate the control knowledge from the domain knowledge (cont.)

- Each rule is given a control pattern (phase xxx) that indicates its applicable phase
- All control rules are then written to transfer control between the different phases

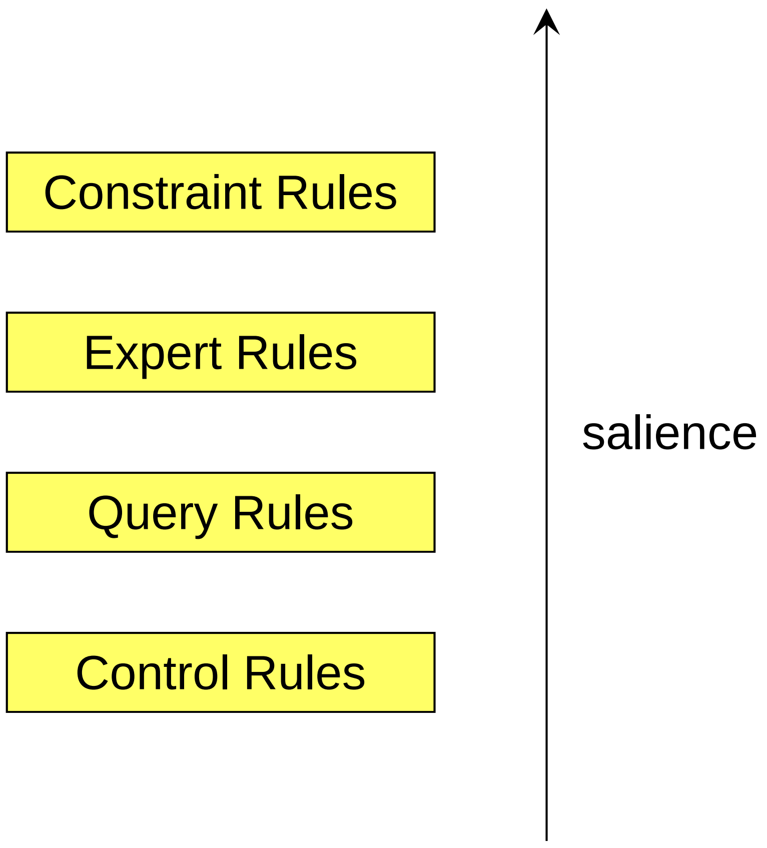
```
(defrule detection-to-isolation
    (declare (salience -10))
    ?phase <- (phase detection)
    =>
     (retract ?phase)
     (assert (phase isolation)))
```

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

- the salience of domain knowledge rules is higher than control rules
  - after all activated rules in one phase are fired, the control rule (i.e. with lower salience) can then be fired (i.e. transfer to next phase)

Salience Hierarchy

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



Using deffacts to control flows

- (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)))

As 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)))

Misuse of Salience

- Because salience is such a powerful tool, allowing explicit control over execution, it can easily be misused.
- Overuse of salience results in a poorly coded program
  - the main advantage of a rule-based program is that the programmer does not have to worry about controlling execution
- Salience should be used to determine the order when rules fire, not for selecting a single rule from a group of rules when patterns can control criteria for selection.

Misuse of Salience

- In a tic-tac-toe game, rules are listed below:
- If a winning square is open, THEN take it
- If a blocking square is open, THEN take it
- If a square is open, THEN take it.
- (defrule pick-to-win
-   (decalre (salience 10))
-   ?phase <- (choose-move)
-   (open-square win)
- =>
-   (retract ?phase)
-   (assert (move-to win)))

- (defrule pick-to-block
- (decalre (salience 5))
-   ?phase <- (choose-move)
-   (open-square block)
- =>
-   (retract ?phase)
-   (assert (move-to block)))

- (defrule pick-any
- ?phase <- (choose-move)
-   (open-square ?any&corner | middle|side)
-   (open-square block)
- =>
-   (retract ?phase)
-   (assert (move-to any)))

Rule of Thumb

- No more than seven salience values should ever be required for coding an expert system – bested limited to 3 – 4.
- For large expert systems, programmers should use modules to control the flow of execution – limited to 2 – 3 salience values.

The Defmodule Construct

- Up to now, all *defrules*, *deftemplates*, and *deffacts* have been contained in a single work space (i.e. MAIN module)
- CLIPS uses the *defmodule* construct to partition a knowledge base by defining the various modules.
- Syntax:

```
(defmodule <module-name> [<comment>])
```

The MAIN Module

By default, CLIPS defines a MAIN module as seen below:

```
CLIPS> (clear)↵
CLIPS> (deftemplate sensor (slot name))↵
CLIPS> (ppdeftemplate sensor)↵
 (deftemplate MAIN::sensor (slot name))
CLIPS>
```

The symbol :: is called the module separator

Examples of Defining Modules

- Examples:

```
CLIPS> (defmodule DETECTION) ↵
CLIPS> (defmodule ISOLATION) ↵
CLIPS> (defmodule RECOVERY)  ↵
```

- the *current* module
  - CLIPS commands operating on a construct work only on the constructs contained in the current module.
  - When CLIPS starts or is cleared, the current module is MAIN
  - When a new modules is defined, it becomes current.
  - The function *set-current-module* is used to change the current module
    - (set-current-module ISOLATION)

Specified Module Name

- The definition of templates and rules would be place in the current module
- To override this, the module where the construct will be placed can be specified in the construct’s name:

```
CLIPS> (defrule ISOLATION::example2 =>) ↵
CLIPS> (ppdefrule example2) ↵
(defrule ISOLATION::example2 =>)
CLIP>
```

Functions about Modules

- To find out which module is current:

```
CLIPS> (get-current-module) ↵
```

- To change the current module:

```
CLIPS> (set-current-module DETECTION) ↵
```

- Specifying modules in commands:

```
CLIPS> (list-defrules RECOVERY) ↵
CLIPS> (list-deffacts ISOLATION) ↵
CLIPS> (list-deftemplates *) ↵
```

  - * indicates all of the modules

Facts in Different Modules

- Just as constructs can be partitioned by placing them in separate modules, facts can also be partitioned.
- Asserted facts are automatically associated with the module in which their corresponding deftemplates are defined.
- The *facts* command can accept a module name as an argument:

```
(facts [<module-name>])
```

Importing/Exporting Facts

- Unlike defrule and deffacts constructs, deftemplate constructs can be shared with other modules.
- A fact is “owned” by the module in which its deftemplate is contained.
- The owning module can export the deftemplate associated with the fact making that fact and all other facts using that deftemplate visible to other modules.

Importing/Exporting Facts

- It is not sufficient just to export the deftemplate to make a fact visible to another module.
- To use a deftemplate defined in another module, a module must also import the deftemplate definition.
- A construct must be defined before it can be specified in an import list, but it does not have to be defined before it can be specified in an export list; so, it is impossible for two modules to import from each other.

Exporting Constructs

- (defmodule DETECTION (export ?ALL))
  - the DETECTION module will export all exportable constructs (e.g. deftemplates and other procedural or OOP constructs)
- (defmodule DETECTION (export ?NONE))
  - the DETECTION module doesn’t export any construct (default state)
- (defmodule DETECTION (export deftemplate ?ALL))
  - the DETECTION module will export all exportable deftemplates
- (defmodule DETECTION (export deftemplate <DT-name>+))
  - the DETECTION module will export a specific list of deftemplates

Importing Constructs

- (defmodule RECOVERY (import DETECTION ?ALL))
  - the RECOVERY module will import all constructs exported from the DETECTION module
- (defmodule RECOVERY (import DETECTION deftemplate ?ALL))
  - the RECOVERY module will import all deftemplates exported from the DETECTION module
- (defmodule RECOVERY (import DETECTION deftemplate <DT-name>+))
  - the RECOVERY module will import a specific list of deftemplates exported from the DETECTION module
- (defmodule RECOVERY (import DETECTION ?ALL) (import ISOLATION deftemplate ?ALL))
  - the RECOVERY module will import constructs from two modules

Example

- (defmodule DETECTION
  - (export deftemplate fault))
- (deftemplate DETECTION::fault
  - (slot component))
- (defmodule ISOLATION
  - (export deftemplate possible-failure))
- (deftemplate ISOLATION::possible-failure
  - (slot component))
- (defmodule RECOVERY
  - (import DETECTION deftemplate fault)
  - (import ISOLATION deftemplate possible-failure))
- CLIPS> (deffacts DETECTION::start (fault (component A)))
- CLIPS> (deffacts ISOLATION::start (possible-failure (component B)))
- CLIPS> (deffacts RECOVERY::start (fault (component C))
- (possible-failure (component D)))
- CLIPS>(reset)
- Try to list facts in these three module: DETECTION, ISOLATION, and RECOVERY

Modules and Execution Control

- The defmodule construct can be used to control the execution of rules.
- Each module defined in CLIPS has its own agenda.
- Execution can be controlled by deciding which module's agenda is selected for executing rules.

The Agenda Command

- To display activations for the current module:

```
CLIPS> (agenda) ↵
```

- To display activations for the DETECTION module:

```
CLIPS> (agenda DETECTION) ↵
```

Example

- (defrule DETECTION::rule-1
  - (fault (component A | C))
  - =>)
- (defrule ISOLATION::rule-2
  - (possible-failure (component B | D))
  - =>)
- (defrule RECOVERY::rule-3
  - (fault (component A | C))
  - (possible-failure (component B | D))
  - =>)
- (get-current-module) ➔ which module now
- (agenda) ➔ check the agenda
- (agenda *) ➔ check all agenda
- If we issue the command (run), what will happen? (i.e. use (watch rules))

The Focus Command

- Assume there are rules on several agendas – when the *run* command is issued, only rules in the current focus module are fired.
- CLIPS maintains a current focus that determines which agenda the run command uses during execution.
- The *reset* and *clear* commands automatically set the current focus to the MAIN module.
- The current focus does not change when the current module is changed.
  - (set-current-module DETECTION) ↵ doesn't change the current focus

The Focus Command

- To change the current focus:

```
CLIPS> (focus <module-name>+)↵
```

- Example:

```
CLIPS> (focus DETECTION) ↵
TRUE
CLIPS> (run)↵
```

- Now rules on the DETECTION module will be fired.

The Focus Command

- The *focus* command not only changes the current focus but also recalls the previous value of the current focus – from the top of the stack data structure called the *focus stack*.
- When the *focus* command changes the current focus, it is pushing the new current focus onto the top of the stack.
- As rules execute, the current focus becomes empty, is popped from the focus stack, the next module becomes the current focus until empty.
- Rules will continue to execute until there are no modules left on the focus stack.

Manipulating the Focus Stack

CLIPS provides several commands for manipulating the current focus and stack:

1. (clear-focus-stack) – removes all modules from focus stack
2. (get-focus) – returns module name of current focus or FALSE if empty
3. (pop-focus) – removes current focus from stack or FALSE if empty

Manipulating the Focus Stack

4. (get-focus-stack) – returns a multifield value containing the modules on the focus stack
5. (watch focus) – can be used to see changes in the focus stack
6. (return) – terminate execution of a rule’s RHS, remove current focus from focus stack, return execution to next module
7. (halt) – terminate the execution of all modules

The MAIN module & run command

If run command is given and focus stack is empty, the MAIN module is automatically pushed onto the focus stack.

```
CLIPS> (clear)
CLIPS> (watch focus)
CLIPS> (watch rules)
CLIPS> (defrule example-1 =>)
CLIPS> (reset)
<== Focus MAIN
==>Focus MAIN
CLIPS>(run)
FIRE          1  example-1: f-0
<==FOCUS MAIN
CLIPS>(defrule example-2 =>)
CLIPS>(agenda)
•  0          example-2: f-0
•  For a total of 1 activation.
•  CLIPS>(list-focus-stack)
•  CLIPS>
```

- The rule *example-2* is on the agenda, but there are no modules on the focus stack. Issuing a *run* command, what happened?
- CLIPS>(run)

Auto-Focus

```
CLIPS> (clear)
CLIPS> (defmodule MAIN (export deftemplate initial-fact))
CLIPS> (defmodule DETECTION (import MAIN deftemplate initial-fact))
CLIPS>(defrule DETECTION::example-1 (declare (auto-focus TRUE)) =>)
CLIPS>(watch focus)
CLIPS>(reset)
<== Focus MAIN
==> Focus MAIN
==> Focus DETECTION from MAIN
CLIPS>
```

- When the *reset* command is issued, the MAIN module is automatically focused on because the focus stack is empty.
- The *example* rule is activated by the assertion of the *initial-fact* fact.
- Since the auto-focus is enabled, the DETECTION module is automatically pushed onto the focus stack.

Implementing the Flow of Control
4. Use modules to organize the rules

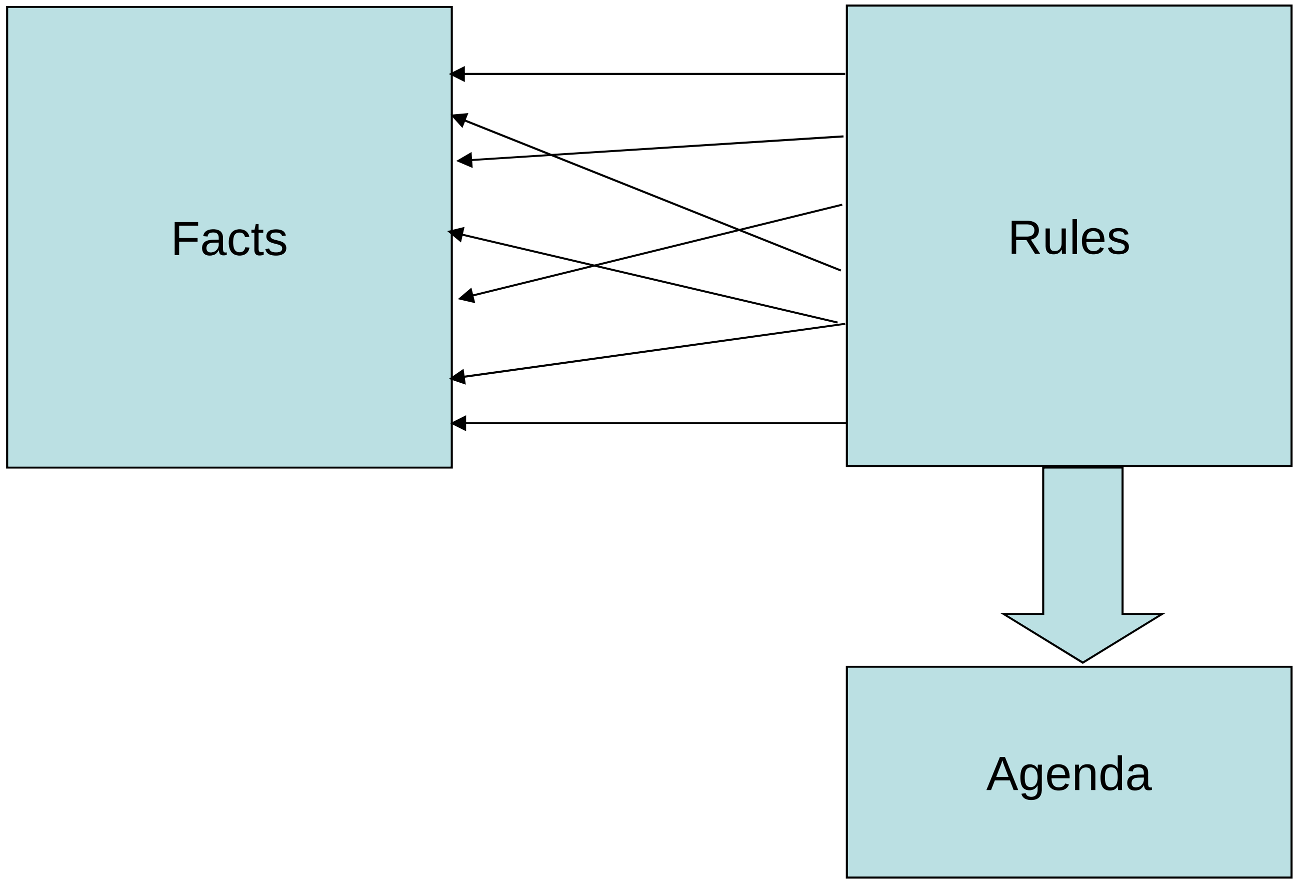
```
(defmodule DETECTION)
(defmodule ISOLATION)
(defmodule RECOVERY)

(deffacts MAIN::control-information
 (phase-sequence DETECTION ISOLATION RECOVERY))
(defrule MAIN::change-phase
  ?list <- (phase-sequence ?next-phase $?other-phases)
    =>
  (focus ?next-phase
  (retract ?list)
  (assert (phase-sequence ?other-phases ?next-phase)))
```

The Rete Pattern-Matching Algorithm

- Rule-based languages like CLIPS use the Rete Pattern-Matching Algorithm for matching facts against the patterns in rules to determine which rules have had their conditions satisfied.
- If the matching process occurs only once, the inference engine simply examines each rule and then searches the set of facts to see if the rule’s patterns have been satisfied – if so it is place on the agenda. (see next slide)

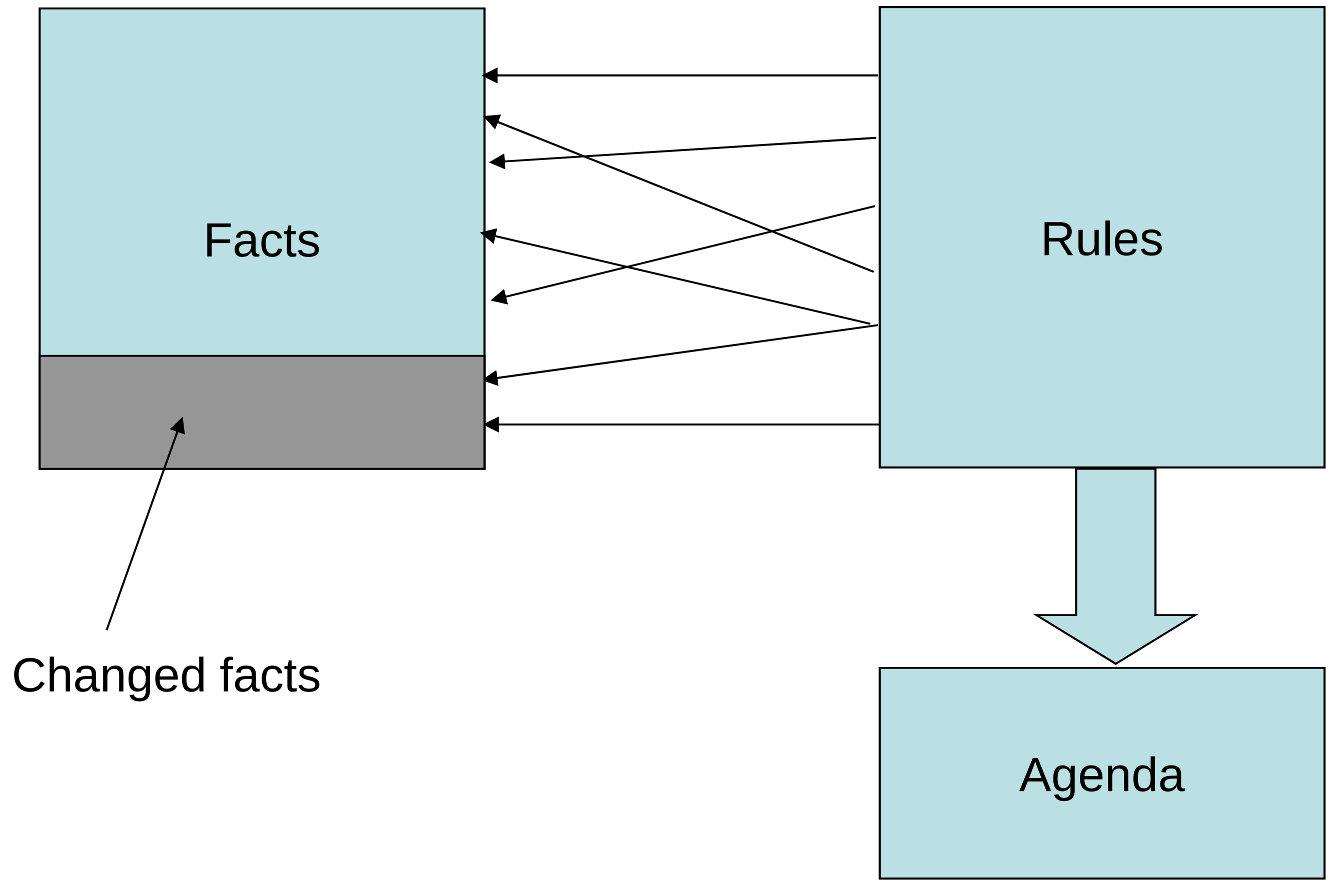
Rules Searching for Facts



Rete Algorithm

- In rule-based languages, the matching process takes place repeatedly and the fact list is modified on each cycle of the execution.
- Such changes can cause previously unsatisfied patterns to be satisfied or vice versa – as facts are added/removed, the set of rules must be updated/maintained.
- Checking rules against facts each cycle is a slow process. (see next slide)

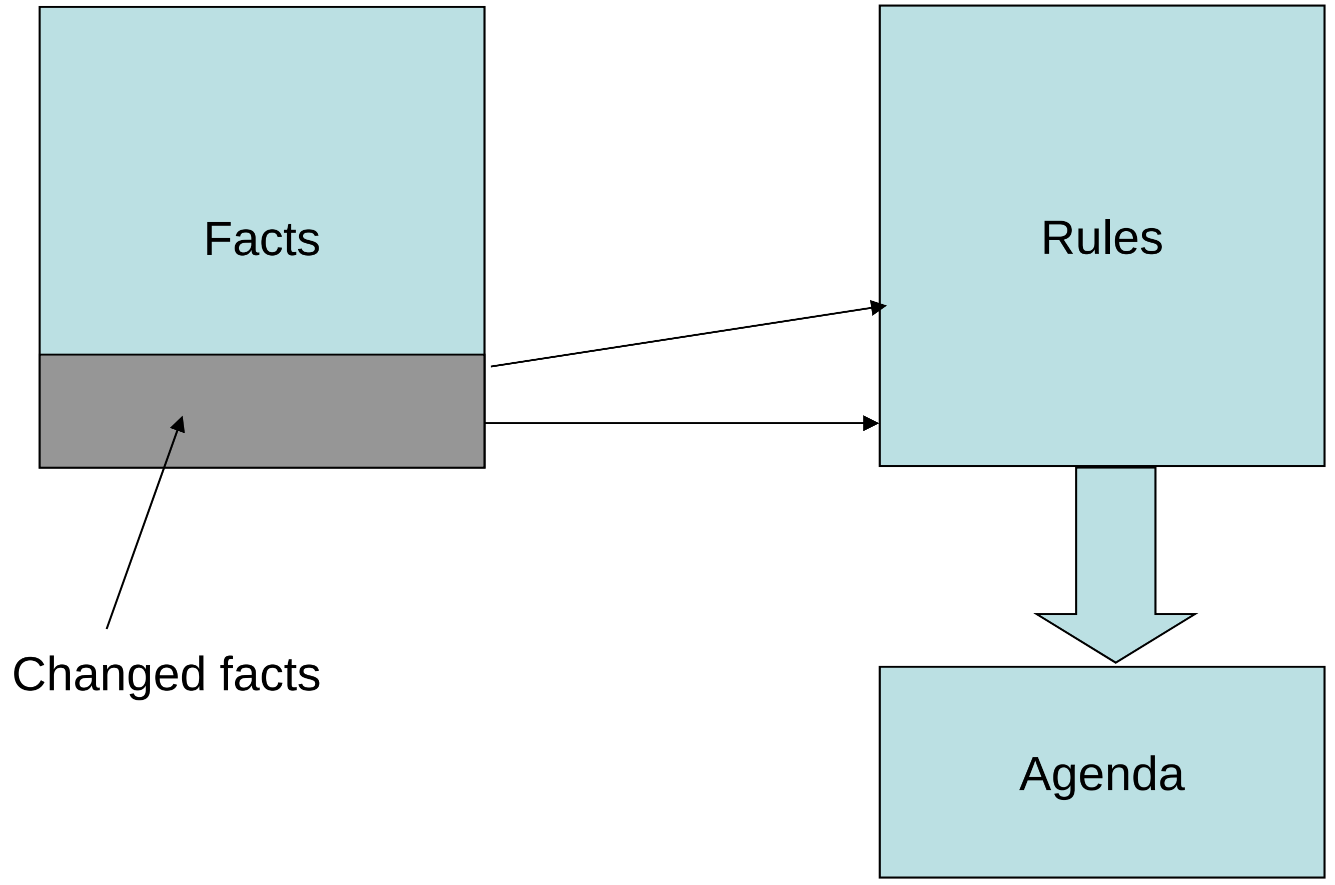
Unnecessary computations when rules search for facts



Rete Algorithm

- Unnecessary computation can be avoided by remembering what has already been matched from cycle to cycle and computing only necessary changes.
- The Rete algorithm uses temporal redundancy to save the state of the matching process from cycle to cycle, recomputing changes in this state only for the change that occurred in the fact list.

Facts searching for Rules



Types of matches

- Partial match
  - Any set of facts that satisfy the rule’s patterns, beginning with the first pattern of the rule and ending with any pattern up to and including the last.
    - A rule with three patterns would have partial matches for the first pattern, the first and second patterns, and the first, second and third patterns.
- Pattern match
  - A pattern match occurs when a fact has satisfied a single pattern in any rule without regard to variables in other patterns that may restrict the matching process

Pattern match

- (data (x 27))
  - The slot x value is equal to the constant 27
- (data (x ~red&~green))
  - The slot x value is not equal to the constant *red* and is not equal to the constant *green*
- (data (x ?x) (y ?y) (z ?x))
  - The z slot value is equal to the x slot value
- (data (x ?x&:(> ?x ?y)))
  - Would not generate a match specification for the x slot because variable ?y is not contained within the pattern.
- (data (x ?x&:(> ?x 4)))
  - The x slot value is greater than the constant 4

Rete Algorithm

- The Rete algorithm also takes advantage of structural similarity in a rules.
- Drawback: the Rete pattern-matching algorithm is memory intensive

Pattern Network

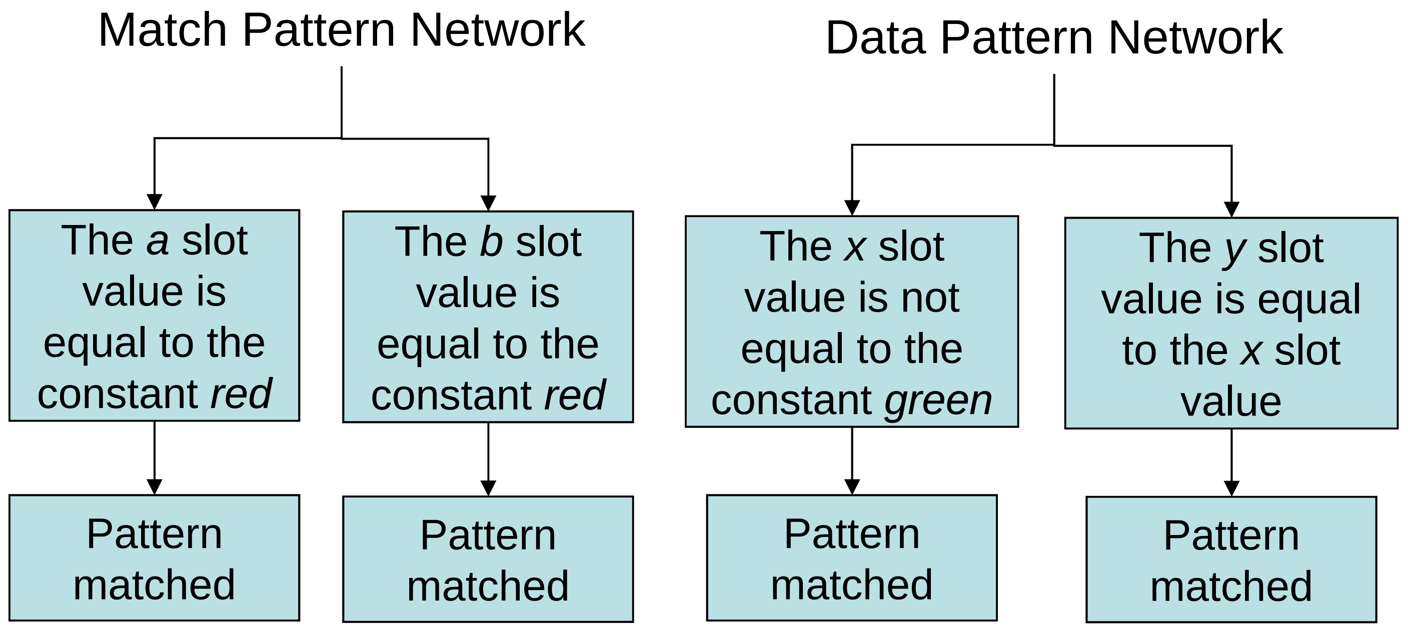
Problems related to matching facts can be divided into two steps:

1. When facts are added and removed it must be determined which patterns have been matched.

2. Comparison of variable bindings across patterns must be checked to determine the partial matches for a group of patterns.

This process is performed in the pattern network – similar to a tree.

Pattern Network for Two Rules


Match Pattern Network
Data Pattern Network
The *a* slot value is equal to the constant *red*
The *b* slot value is equal to the constant *red*
The *x* slot value is not equal to the constant *green*
The *y* slot value is equal to the *x* slot value
Pattern matched
Pattern matched
Pattern matched
Pattern matched

```
(defrule Rete-rule-1
  (match (a red)
  (data (x ?x) (y ?x))
=>)
```

```
(defrule Rete-rule-2
  (match (a ?x) (b red))
  (data (x ~green) (y ?x))
  (data (x ?x) (y ?x))
=>)
```

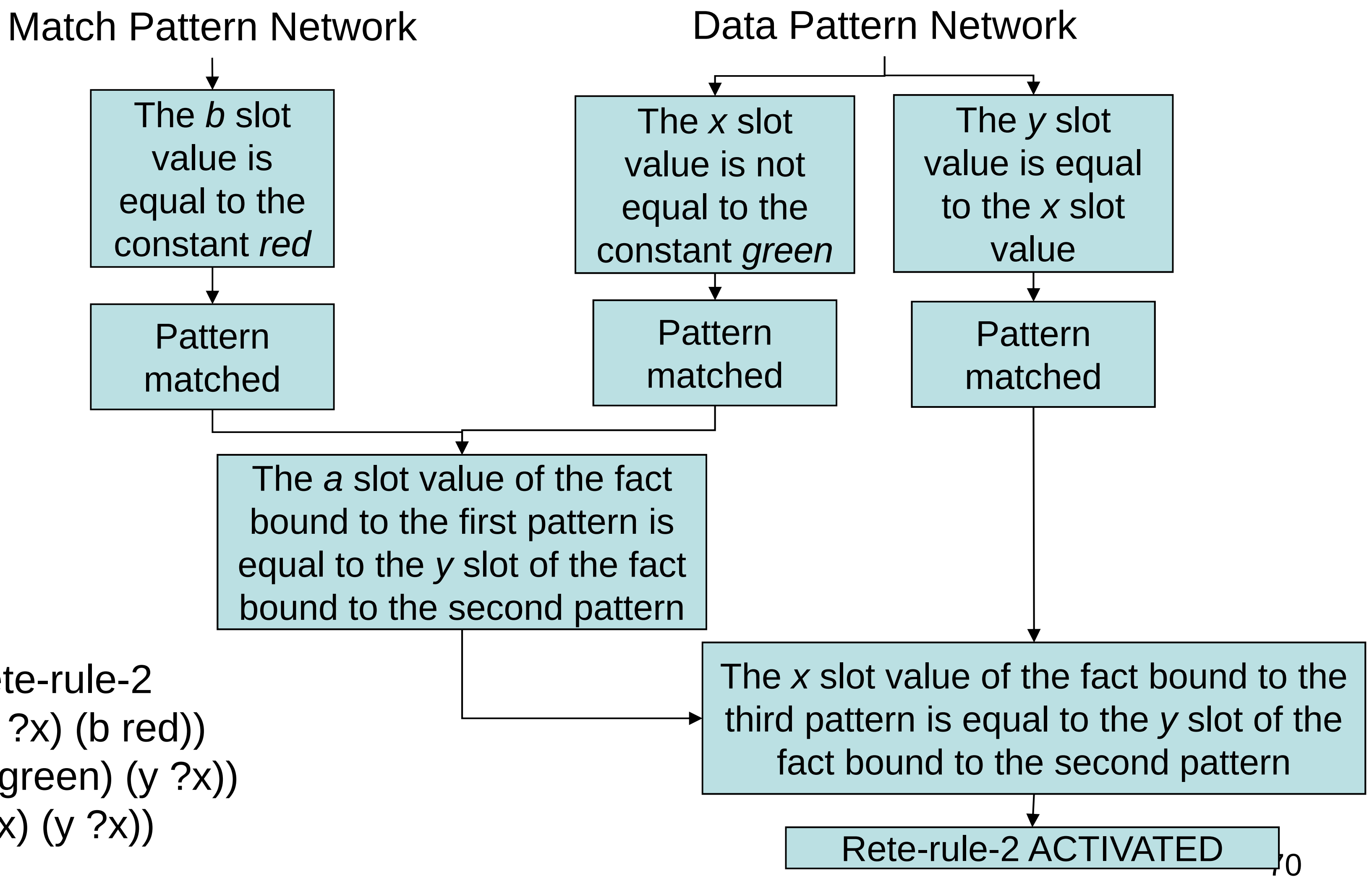
Join Network

- Once it has been determined which patterns have been matched by facts, comparison of variable binding across patterns must be checked to ensure that variables used in more than one pattern have consistent values.
- This is done in the *join network* which takes advantage of structural similarity by sharing joins between rules.

Join Network

- (defrule Rete-rule-2
  - (match (a ?x) (b red))
  - (data (x ~green) (y ?x))
  - (data (x ?x) (y ?x))
  - =>)
- First join
  - The a slot value of the fact bound to the first pattern is equal to the y slot value of the fact bound to the second pattern
- Second join
  - The x slot value of the fact bound to the third pattern is equal to the y slot value of the fact bound to the second pattern

Pattern and Join Network for Rule Rete-Rule-2



Match Pattern Network
Data Pattern Network
The *b* slot value is equal to the constant *red*
The *x* slot value is not equal to the constant *green*
The *y* slot value is equal to the *x* slot value
Pattern matched
Pattern matched
Pattern matched
The *a* slot value of the fact bound to the first pattern is equal to the *y* slot of the fact bound to the second pattern
The *x* slot value of the fact bound to the third pattern is equal to the *y* slot of the fact bound to the second pattern
Rete-rule-2 ACTIVATED

```
(defrule Rete-rule-2
  (match (a ?x) (b red))
  (data (x ~green) (y ?x))
  (data (x ?x) (y ?x))
=>)
```

Structural Similarity

- (defrule sharing-1
  - (match (a ?x) (b red))
  - (data (x ~green) (y ?x))
  - (data (x ?x) (y ?x))
  - (other (q ?z)
  - =>)
- (defrule sharing-2
  - (match (a ?y) (b red))
  - (data (x ~green) (y ?y))
  - (data (x ?y) (y ?y))
  - (other (q ?y))
  - =>)
- The two rules could share all of their pattern network and the joins created for their first ***three patterns*** in the join network.
- The join for the fourth pattern cannot be shared.

Importance of Pattern Order

- Because the Rete algorithm saves the state from one cycle to the next, it is important to make sure that rules do not generate large numbers of partial matches.

Compare the two rules with same deffacts

- Use (matches <rule-name>) to see the matches

```
(deffacts information
  (find-match a c e g)
  (item a)
  (item b)
  (item c)
  (item d)
  (item e)
  (item f)
  (item g))
```

```
(defrule match-1
  (find-match ?x ?y ?z ?w)
  (item ?x)
  (item ?y)
  (item ?z)
  (item ?w)
=>
  (assert (found-match ?x ?y ?z ?w)))
 (defrule match-2  (item ?x)
   (item ?y)
   (item ?z)
   (item ?w)
   (find-match ?x ?y ?z ?w)
 =>
   (assert (found-match ?x ?y ?z ?w)))
```

Counting for Rule match-1

- Pattern matches:
  - Pattern 1: f-1
  - Pattern 2: f-2, f-3, f-4, f-5, f-6, f-7, f-8
  - Pattern 3: f-2, f-3, f-4, f-5, f-6, f-7, f-8
  - Pattern 4: f-2, f-3, f-4, f-5, f-6, f-7, f-8
  - Pattern 5: f-2, f-3, f-4, f-5, f-6, f-7, f-8
- Partial matches
  - Pattern 1: [f-1]
  - Pattern 2: [f-1, f-2]
  - Pattern 3: [f-1, f-2, f-4]
  - Pattern 4: [f-1, f-2, f-4, f-6]
  - Pattern 5: [f-1, f-2, f-4, f-6, f-8]
- 29 pattern matches and 5 partial matches

Fact IDs:
f-1 (find-matc a c e g)
f-2 (item a)
f-3 (item b)
f-4 (item c)
f-5 (item d)
f-6 (item e)
f-7 (item f)
f-8 (item g)

Counting for Rule match-2

- Pattern matches:
  - Pattern 1: f-2, f-3, f-4, f-5, f-6, f-7, f-8
  - Pattern 2: f-2, f-3, f-4, f-5, f-6, f-7, f-8
  - Pattern 3: f-2, f-3, f-4, f-5, f-6, f-7, f-8
  - Pattern 4: f-2, f-3, f-4, f-5, f-6, f-7, f-8
  - Pattern 5: f-1
- Partial matches
  - Pattern 1: [f-2], [f-3], [f-4], [f-5], [f-6], [f-7], [f-8]
  - Pattern 2: [f-2, f-2], [f-2, f-3], [f-2, f-4], [f-2, f-5]… => 49 partial matches
  - Pattern 1 through 4: 2401 partial matches
  - Pattern 1 through 5: only one partial match
- 29 pattern matches and 2801 partial matches

Fact IDs:
f-1 (find-matc a c e g)
f-2 (item a)
f-3 (item b)
f-4 (item c)
f-5 (item d)
f-6 (item e)
f-7 (item f)
f-8 (item g)

Guidelines for Ordering Patterns

Guidelines:

- Place the most specific pattern toward the front of the LHS of a rule.
- Patterns matching against facts frequently added/removed from fact list should be placed toward the end of the LHS of a rule.
- Patterns that will match very few facts in the fact list should be placed near the front of the rule.

General Rules vs. Specific Rules

1. Specific rules tend to isolate much of the pattern-matching process in the pattern network, reducing the amount of work in the join network.
2. General rules often provide more opportunity for sharing in the pattern and join networks.
3. A single rule can also be more maintainable than a group.
4. General rules need to be written carefully.

Example: Specific vs. General

```
(deftemplate location (slot x) (slot y))
(defrule move-north
  (move north)
  ?old-location <- (location (y ?old-y))
=>
  (modify ?old-location (y (+ ?old-y 1))))
(defrule move-south
  (move south)
  ?old-location <- (location (y ?old-y))
=>
  (modify ?old-location (y (- ?old-y 1))))
(defrule move-east
  (move east)
  ?old-location <- (location (x ?old-x))
=>
  (modify ?old-location (x (+ ?old-x 1))))
(defrule move-west
  (move west)
  ?old-location <- (location (x ?old-x))
=>
  (modify ?old-location (x (- ?old-x 1))))
```

```
(deftemplate direction (slot which-way)
                                    (slot delta-x)
                                    (slot delta-y))
(deffacts direction-information
  (direction (which-way north)
                (delta-x 0) (delta-y 1))
 (direction (which-way south)
                (delta-x 0) (delta-y -1))
 (direction (which-way east)
                (delta-x 1) (delta-y 0))
 (direction (which-way west)
                (delta-x -1) (delta-y 0)))
(defrule move-direction
  (move ?dir)
  (direction (which-way ?dir) (delta-x ?dx)
                 (delta-y ?dy))
  ?old-location <- (location (x ?old-x) (y ?old-y))
=>
  (modify ?old-location (x (+ ?old-x ?dx)
                                     (y (+ ?old-y dy))))
```

Simple Rules vs. Complex Rules

1. The easiest way to code a problem in a rule-based language is not necessarily the best.
   1. e.g. find the largest number by using not CE

2. The number of comparisons performed can often be reduced by using temporary facts to store data.

Simple Rules vs. Complex Rules

- Simple

```
(defrule largest-number
  (number ?number1)
  (not (number ?number2&: (> number2 ?number1)))
=>
  (printout t “Largest number is “ ?number crlf))
```

- Complex

```
(defrule try-number
  (number ?n)
=>
  (assert (try-number ?n)))
(defrule largest-unknown
  ?attempt <- (try-number ?n)
  (not (largest ?))
=>
  (retract ?attempt)
  (assert (largest ?n)))
```

```
(defrule largest-smaller
  ?old-largest <- (largest ?n1)
  ?attempt <- (try-number ?n2&:(> ?n2 ?n1))
=>
  (retract ?old-largest ?attempt)
  (assert (largest /n2)))
(defrule largest-bigger
  (largest ?n1)
  ?attempt <- (try-number ?n2&: (<= ?n2 ?n1))
=>
  (retract ?attempt))
```

```
(defrule print-largest
  (declare (salience -1))
  (largest ?number)
=>
  (printout t “Largest number is “ ?number crlf))
```

CLIPS Commands - Math

1. (max <numeric>+)
   - returns the value of its largest argument
   - (max 10 5 15 12) $\rightarrow$ 15
2. (min <numeric>+)
   - returns the value of its smallest argument
   - (min 10 5 15 12) $\rightarrow$ 5
3. (** <numeric-1> <numeric-2>)
   - returns the value of <numeric-1> raised to the power of <numeric-2>
   - (** 2 3) $\rightarrow$ 8
4. (round <numeric>)
   - returns the value of argument rounded to the closest integer
   - (round 6.6) $\rightarrow$ 7

CLIPS Commands - Multifield

1. (create$ <expression>+)
   - create a multifield value
   - (create$ a b c d) ➔ (a b c d)
2. (explode$ <string>)
   - returns a multifield value from a string
   - (explode$ “a b c d”) ➔ (a b c d)
3. (implode$ <multifield>)
   - returns a string containing the fields from a multifield value
   - (implode$ (create$ a b c d)) ➔ “a b c d”

CLIPS Commands – Multifield (cont.)

4. (first$ <multifield>)
   - returns the first field of <multifield>
   - (first$ (create$ a b c d)) ➔ a
5. (rest$ <multifield>)
   - returns a multifield value containing all but the first field
   - (rest$ (create$ a b c d)) ➔ (b c d)
6. (nth$ <integer> <multifield>)
   - returns the ***n***th field containing in <multifield>
   - (nth$ 3 (create$ a b c d)) ➔ c

CLIPS Commands – Multifield (cont.)

7. (member$ <single-field> <multifield>)
   - examines if the <single-field> is in the <multifield> or not
   - returns the position if in the <multifield> or FALSE if not
   - (member$ b (create$ a b c d)) ➔ 2
8. (length$ <multifield>)
   - returns the number of fields in a multifield value
   - (length$ (create$ a b c d)) ➔ 4
9. (subseq$ <multifield> <begin> <end>)
   - extracts the fields in the specified range and returns them in a multifield value
   - (subseq$ (create$ a b c d e) 2 4) ➔ (b c d)
10. (subsetp <multifield-1> <multifield-2>)
    - returns TRUE in <multifield-1> is a subset of <multifield-2>, otherwise FALSE
    - (subsetp (create$ b c d) (create$ a b c d e)) ➔ TRUE

CLIPS Commands – Multifield (cont.)

11. (insert$ <multifield> <integer> <single-or-multifield>)
    - inserts the <single-or-multifield> vaule into the <multifield>, before the ***n***th value
    - (insert$ (create$ a b c d) 3 h) ➔ (a b h c d)
12. (delete$ <multifield> <begin> <end>)
    - deletes the fields in the specified range and returns the result
    - (delete$ (create$ a b c d e) 2 4) ➔ (a e)
13. (delete-member$ <multifield> <single-or-multifield>+)
    - deletes specified values contained within a multifield value
    - (delete-member$ (create$ a b c d e f g) (create$ b c) f e) ➔ (a d g)

CLIPS Commands – Multifield (cont.)

14. (replace$ <multifield> <begin> <end> <single-or-multifield>)
    - replaces the fields in the specified range with the <sigle-or-multifield> value and returns the result
    - (replace$ (create$ a b c d e) 2 4 (create$ f g)) ➔ (a f g e)
15. (replace-member$ <multifield> <substitute> <search>+)
    - replaces specified values contained within a multifield value
    - (replace-member$ (create$ a b c d e f g h) k g (create$ b c) e) ➔ (a k d k f k h)

CLIPS Commands – Predicate

1. (eq <expression> <expression>+)
   - returns TRUE if its first argument is equal in type and value to all its subsequent arguments, otherwise FALSE
   - (eq (create$ a b c) (explode$ “a b c”)) ➔ TRUE
   - (eq 5 5.0) ➔ FALSE
2. (neq <expression> <expression>+)
   - returns TRUE if its first argument is not equal in type and value to all its subsequent arguments, otherwise FALSE
   - (neq (implode$ (create$ a b c)) “a b c”) ➔ FALSE
3. logical commands
   - (and <expression>+) , (or <expression>+) , (not <expression>)
   - (= <numeric-1> <numeric-2>) , (<> <numeric-1> <numeric-2>) , (> <numeric-1> <numeric-2>) , (< <numeric-1> <numeric-2>) , (>= <numeric-1> <numeric-2>) , (<= <numeric-1> <numeric-2>)

CLIPS Commands – Predicate (cont.)

4. (multifieldp <expression>)
    - returns TRUE if <expression> is a multifield value, otherwise FALSE
5. data type testing
    - (numberp <expression>) : a float or an integer
    - (integerp <expression>)
    - (floatp <expression>)
    - (lexemep <expression>) : a string or a symbol
    - (stringp <expression>)
    - (symbolp <expression>)