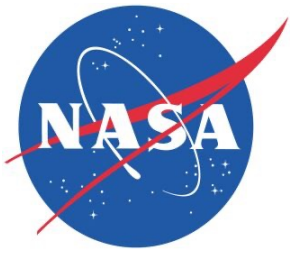
Artificial Intelligence

Xiaoqing Zheng
zhengxq@fudan.edu.cn

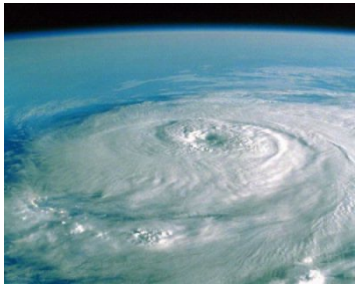
Programming language

CLIPS

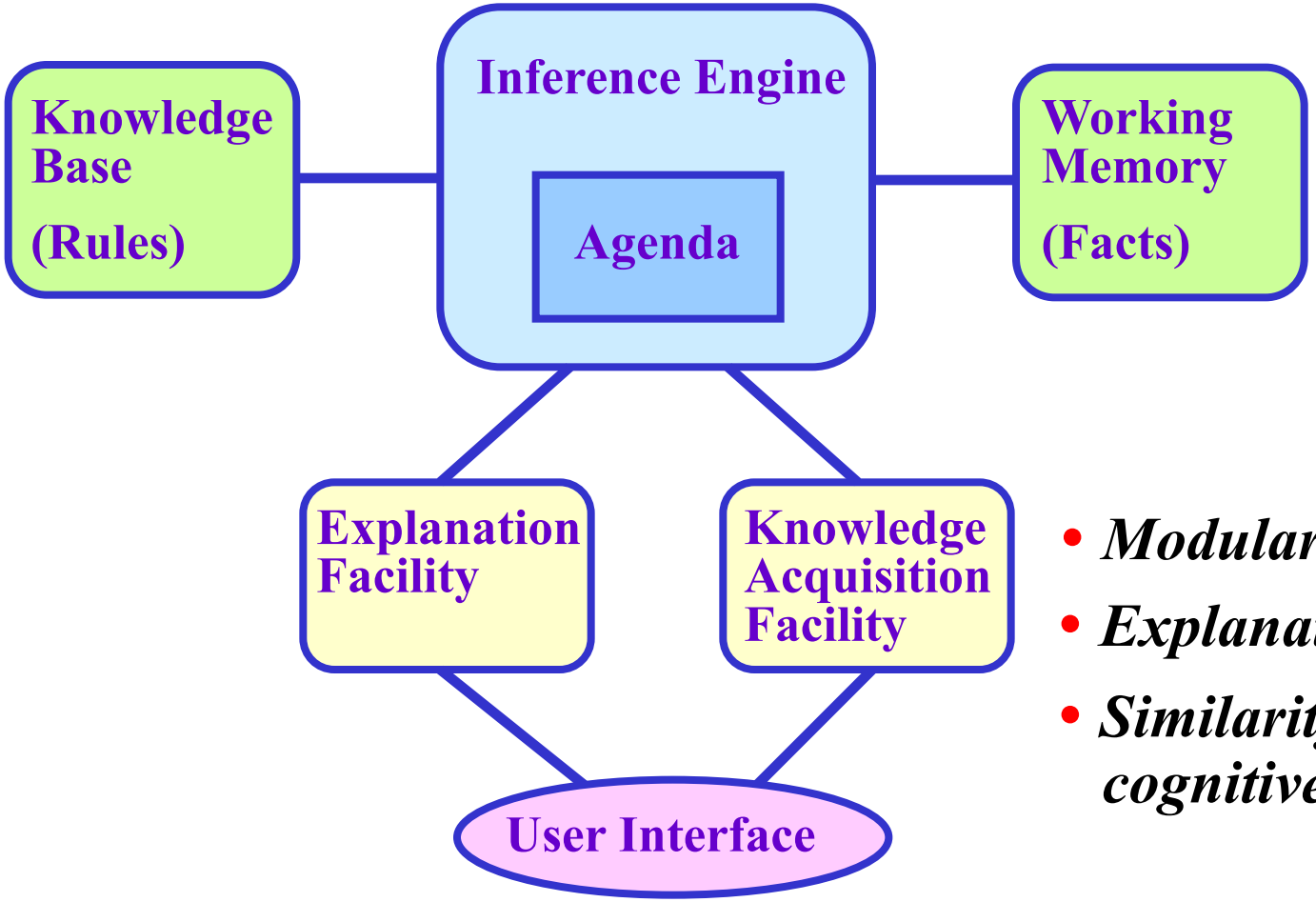
C language Integrated Production System



National Aeronautics and Space Administration



Rule-based expert system structure



- ***Modular nature.***
- ***Explanation facilities.***
- ***Similarity to human cognitive process.***

Basic components

- Fact list
- knowledge base
- Inference engine

Notation

( )	basic delimiters	(initial-fact)
< >	replacement	<integer>
[ ]	optional	[<integer>]
+	one or more	<float>+
*	zero or more	<integer>*
|	choice	all | none | some

Primitive data types

integer	1, +3, –1, 65
float	1.5, 0.7, 9e+1, 3.5e10
symbol	fire, 345B, activate-sprinkler-system
string	"John Smith", "string", " string"

Delimiter:

spaces, tabs, carriage returns, line feeds,
", (,), ;, &, |, ~, <, >

? and $? cannot be placed at the beginning of a symbol since they are used to denote variables.

Facts

Fact example

```
(person (name "John Smith")
        (age 21)
        (eye-color black)
        (hair-color black))
```

Template

```
(deftemplate person "A person template"
        (slot name)
        (slot age)
        (slot eye-color)
        (slot hair-color))
```

deftemplate

deftemplate syntax

(deftemplate <relation-name>
 [<optional-comment> <slot-definition>*])

<slot-definition> is defined as
(slot <slot-name> <attributes>*) |
(multislot <slot-name><attributes>*)

multislot example

(multislot values)

(values 7 9 3 4 20)

deftemplate type attributes

Type attribute

(type <type-specification>)

<type-specification> is one of
?VARIABLE (default)
?SYMBOL
?STRING
?LEXEME (?SYMBOL or ?STRING)
?INTEGER
?FLOAT
?NUMBER (?INTEGER or ?FLOAT)

Example:

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

deftemplate allowed value attributes

Allowed value attribute

(<allowed-value> <value>+)

<allowed-value> is one of
?allowed-values ?VARIABLE (default)
?allowed-values
?allowed-symbols
?allowed-strings
?allowed-lexemes
?allowed-integers
?allowed-floats
?allowed-numbers

Example:

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

deftemplate range attributes

Range attribute

(range <lower-limit> <upper-limit>)

<lower-limit> <upper-limit> is one of
?VARIABLE (default)
<number>

Example:

```
(deftemplate person
  (multislot name (type SYMBOL))
  (slot age (type INTEGER) (range 0 ?VARIABLE))
  (slot gender (allowed-values male female)))
```

deftemplate cardinality attributes

Cardinality attribute

(cardinality <lower-limit> <upper-limit>)

<lower-limit> <upper-limit> is one of
?VARIABLE (default)
positive integer

Example:

```
(deftemplate person
  (multislot name (type SYMBOL) (cardinality 1 6))
  (slot age (type INTEGER) (range 0 ?VARIABLE))
  (slot gender (allowed-values male female)))
```

deftemplate default attributes

Default attribute

(default <default-specification>)

<default-specification> is one of
?DERIVE (default)
?NONE
a single expression (for a single-field slot)
zero or more expressions (for a multifield slot)

Example:

```
(deftemplate person
  (multislot name (type SYMBOL) (cardinality 1 6))
  (slot age (type INTEGER) (range 0 ?VARIABLE))
  (slot gender (allowed-values male female) (default female)))
```

deftemplate default attributes

```
(deftemplate default-derive-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 10.0)))

(assert (default-derive-example))

(default-derive-example (a  nil)
                        (b  0)
                        (c  red)
                        (d      )
                        (e  3.5 3.5))
```

deftemplate attributes

Attributes	Syntax
type	(type <type-specification>)
allowed value	(<allowed-value> <value>+)
range	(range <lower-limit> <upper-limit>)
cardinality	(cardinality <lower-limit> <upper-limit>)
default	(default <default-specification>)

Ordered facts

Ordered fact example

(number-list 7 9 3 4 20)

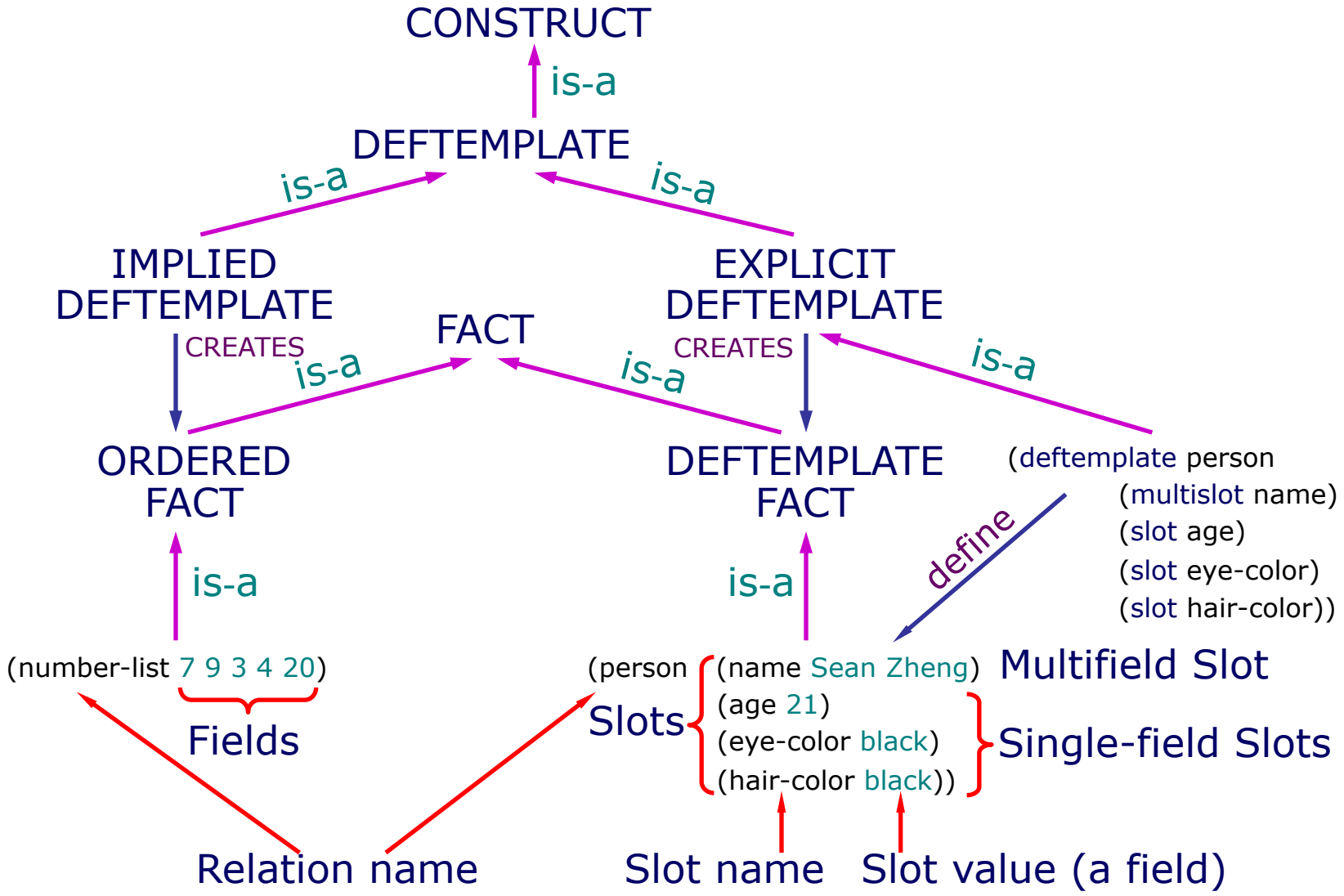
This is equivalent to

(deftemplate number-list (multislot values))
(number-list ((values 7 9 3 4 20))

Some examples

(initial-fact)
(time 9:55)

deftemplate



Fact operations

Add facts

(assert <fact>+)

Display facts

(fact [<start> [<end> [<maximum>]]])

Remove facts

(retract <fact-index>+)

Modify facts

(modify <fact-index> <slot-modifier>+)

<slot-modifier> is defined as
(<slot-name> <slot-value>)

deffacts

deffacts syntax

```
(deffacts <deffacts-name> [<optional-comment>]
 <facts>*)
```

deffacts example

```
(deffacts people "Some people we know"
    person (name "Sean Smith") (age 21)
           (eye-color black) (hair-color black))
    person (name "John Riley") (age 24)
           (eye-color blue) (hair-color black))
    person (name "Bruce Lenny") (age 35)
           (eye-color blue) (hair-color blond)))
```

Rule

if the emergency is a fire

then the response is to activate
the sprinkler system

```
(deftemplate emergency (slot type))
(deftemplate response (slot action))

(defrule fire-emergency
    (emergency (type fire))
    =>
    (assert (response
              (action activate-sprinkler-system))))
```

defrule and run

defrule syntax

(defrule <rule-name> [<optional-comment>]
    <patterns>* ; Left-Hand Side (LHS) of the rule
    =>
    <actions>* ; Right-Hand Side (RHS) of the rule

Run syntax

(run [<limit>])

Manipulating constructs

Display constructs

(list-defrules) (list-deftemplates) (list-deffacts)

Pretty print constructs

(ppdefrule <defrule-name>)
(ppdeftemplate <deftemplate-name>)
(ppdeffacts <deffacts-name>)

Delete constructs

(undefrule <defrule-name>)
(undeftemplate <deftemplate-name>)
(undeffacts <deffacts-name>)

printout and clear

printout syntax

(printout <logical-name> <print-items>*)

printout example

```
(defrule fire-emergency
    (emergency (type fire))
    =>
    (printout t "Activate the sprinkler system" crlf))
```

clear syntax

(clear)

Math function

prefix form

(+ 2 3)
(* 4.2 5)

3 + 4 × 5

(+ 3 (* 4 5))

(y2 – y1) / (x2 – x1) > 0

(> (/ (– y2 y1) (– x2 x1)) 0)

Variable

```
(deftemplate person
  (slot name)
  (slot eyes)
  (slot hair))

(defrule find-blue-eyes
  (person (name ?name) (eyes blue))
  =>
  (printout t ?name " has blue eyes." crlf))

(deffacts people
  (person (name Jane) (eyes blue) (hair red))
  (person (name Joe) (eyes green) (hair brown))
  (person (name Jack) (eyes blue) (hair black))
  (person (name Jeff) (eyes green) (hair brown)))
```

Variable

```
(deftemplate find
  (slot eyes))

(defrule find-eyes
  (find (eyes ?eyes))
  (person (name ?name) (eyes ?eyes))
  =>
  (printout t ?name " has " ?eyes " eyes." crlf))
```

```
(assert (find (eyes green)))
(run)
```

Fact address

```
(deftemplate person
  (slot name) (slot address))

(deftemplate moved
  (slot name) (slot address))

(defrule process-moved-information
  ?f1 <- (moved (name ?name) (address ?address))
  ?f2 <- (person (name ?name))
  =>
  (retract ?f1)
  (modify ?f2 (address ?address)))

(deffacts example
  (person (name "John Hill") (address "25 Mulberyy Lane"))
  (moved (name "John Hill") (address "37 Cherry Lane")))
```

Single-field wildcard

```
((deffacts people
  (person (name John Q. Smith)
  (identity-card-number 33010219860520381O))
  (person (name Jack R. Lenny)
  (identity-card-number 3301061985061215 21))

(defrule print-identity-card-number
  (print-id-number-name ?last-name)
  (person (name ?first-name ?middle-name ?last-name)
            (identity-card-number ?id-number))
  =>
  (printout t ?id-number crlf))
```

Single-field wildcard

```
((deffacts people
  (person (name John Q. Smith)
  (identity-card-number 33010219860520381O))
  (person (name Jack R. Lenny)
  (identity-card-number 3301061985061215 21))

(defrule print-identity-card-number
  (print-id-number-name ?last-name)
  (person (name ? ? ?last-name)
            (identity-card-number ?id-number))
  =>
  (printout t ?id-number crlf))
```

Unspecified slot

```
(deftemplate person
  (multislot name)
  (slot identity-card-number))

(person (name John Q. Smith))
```

This pattern is equivalent to

```
(person (name John Q. Smith)
        (identity-card-number ?))
```

Multi-field wildcard

```
(deftemplate person
  (multislot name)
  (multislot children))

(defrule find-child
  (find-child ?child)
  (person (name $?name)
          (children $?before ?child $?after))
  =>
  (printout t ?name " has child " ?child crlf)
  (printout t "Other children are " ?before " " ?after crlf))

(deffacts people
  (person (name John Smith) (children Jane Joe Paul))
  (person (name Jack R. Lenny) (children Joe Joe Joe)))
```

Multi-field wildcard

(Jack R. Lenny) has child Joe
Other children are () (Joe Joe)

(Jack R. Lenny) has child Joe
Other children are (Joe) (Joe)

(Jack R. Lenny) has child Joe
Other children are (Joe Joe) ()

(John Smith) has child Joe
Other children are (Jane) (Paul)

Connective constraint

```
(defrule person-without-black-hair
  (person (name ?name) (hair ~black))
  =>
  (printout t ?name " does not have black hair " crlf))
```

```
(defrule black-or-brown-hair
  (person (name ?name) (hair brown|black))
  =>
  (printout t ?name " has dark hair " crlf))
```

```
(defrule black-or-brown-hair
  (person (name ?name) (hair ?color&brown|black))
  =>
  (printout t ?name " has " ?color " hair " crlf))
```

Combing field constraints

The first person has either blue or green eyes and does not have black hair. The second person does not have the same color eyes as the first person and has either black hair or the same color as the first person.

```
(defrule complex-eye-hair-match
  (person (name ?name1)
          (eyes ?eyes1&blue|green)
          (hair ?hair1&~black))
  (person (name ?name2&~?name1)
          (eyes ?eyes2&~?eyes1)
          (hair ?hair2&black|?hair1))
  =>
  (printout t ?name1 " has " ?eyes1 " eyes and "
                             ?hair1 " hair " crlf)
  (printout t ?name2 " has " ?eyes2 " eyes and "
                             ?hair2 " hair " crlf))
```

Predicate function

and

```
(and (> 4 3) (> 4 5))
FALSE
```

or

```
(or (> 4 3) (> 4 5))
TRUE
```

not

```
(not (integerp 3))
FALSE
```

test

```
(test and (integerp 3)
          (>= 3 1))
TRUE
```

Predicate function

Pattern

```
(age ?age)
(test (> ?age 18))
```

This pattern is equivalent to

```
(age ?age&:(> ?age 18))
```

or conditional element

```
(defrule shut-off-electricity-1
  ?power <- (electrical-power (status on))
  (emergency (type flood))
  =>
  (modify ?power (status off))
  (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
  ?power <- (electrical-power (status on))
  (extinguisher-system (type water-sprinkler)
                       (status on))
  =>
  (modify ?power (status off))
  (printout t "Shut off the electricity" crlf))
```

or conditional element

```
(defrule shut-off-electricity
  ?power <- (electrical-power (status on))
  (or (emergency (type flood))
      (extinguisher-system (type water-sprinkler)
                           (status on)))
  =>
  (modify ?power (status off))
  (printout t "Shut off the electricity" crlf))
```

and conditional element

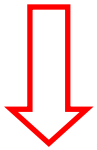
```
(defrule use-carbon-dioxide-extinguisher
  ?system <- (extinguisher-system (type carbon-dioxide)
                                  (status off))
  (or (emergency (type class-B-fire))
      (and (emergency (type class-C-fire))
           (electrical-power (status off))))
  =>
  (modify ?system (status on))
  (printout t "Use carbon dioxide extinguisher" crlf))
```

not conditional element

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

not conditional element

```
(defrule no-birthdays-on-specific-date
  (not (person (birthday ?date)))
  (check-for-no-birthdays (date ?date)
  =>
  (printout t "No birthdays on " ?date crlf))
```



```
(defrule no-birthdays-on-specific-date
  (check-for-no-birthdays (date ?date)
  (not (person (birthday ?date)))
  =>
  (printout t "No birthdays on " ?date crlf))
```

exists conditional element

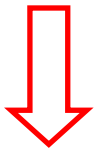
```
(defrule operator-alert-for-emergency
  (emergency)
  =>
  (printout t "Emergency: Operator Alert." crlf)
  (assert (operator-alert)))

(deffacts some-emergency
  (emergency (type fire))
  (emergency (type flood)))
```

```
Emergency: Operator Alert.
Emergency: Operator Alert.
```

exists conditional element

```
(defrule operator-alert-for-emergency
  (exists (emergency))
  =>
  (printout t "Emergency: Operator Alert" crlf)
  (assert (operator-alert)))
```



(initial-fact)

The rule is equivalent to

```
(defrule operator-alert-for-emergency
  (and (not (not (and (emergency)))))
  =>
  (printout t "Emergency: Operator Alert" crlf)
  (assert (operator-alert)))
```

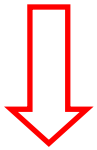
forall conditional element

```
(defrule all-fires-being-handled
  (forall (emergency (type fire)
                     (location ?where))
          (fire-squad (location ?where))
          (evacuated (building ?where)))
  =>
  (printout t "All buildings that are on fire have been" crlf
              "evacuated and have firefighters on locaiton." crlf))

(deffacts locations
  (emergency (type fire) (location lab))
  (fire-squad (name A) (location lab))
  (evacuated (building lab))
  (emergency (type fire) (location sport-hall))
  (fire-squad (name B) (location sport-hall)))
```

forall conditional element

```
(forall <first-CE>
        <remaining-CEs>+)
```



This pattern is equivalent to

```
(not (and <first-CE>
          (not (and <remaining-CES>+))))
```

logical conditional element

```
(defrule use-oxygen-masks
  (emergency (type fire))
  (noxious-fumes-present)
  (gas-extinguishers-in-use)
  =>
  (assert (use-oxygen-masks)))

(deffacts conditions
  (emergency (type fire))
  (noxious-fumes-present)
  (gas-extinguishers-in-use))
```

logical conditional element

```
(defrule use-oxygen-masks
  (logical (noxious-fumes-present)
           (gas-extinguishers-in-use))
  (emergency (type fire))
  =>
  (assert (use-oxygen-masks)))
```

Dependents and dependencies

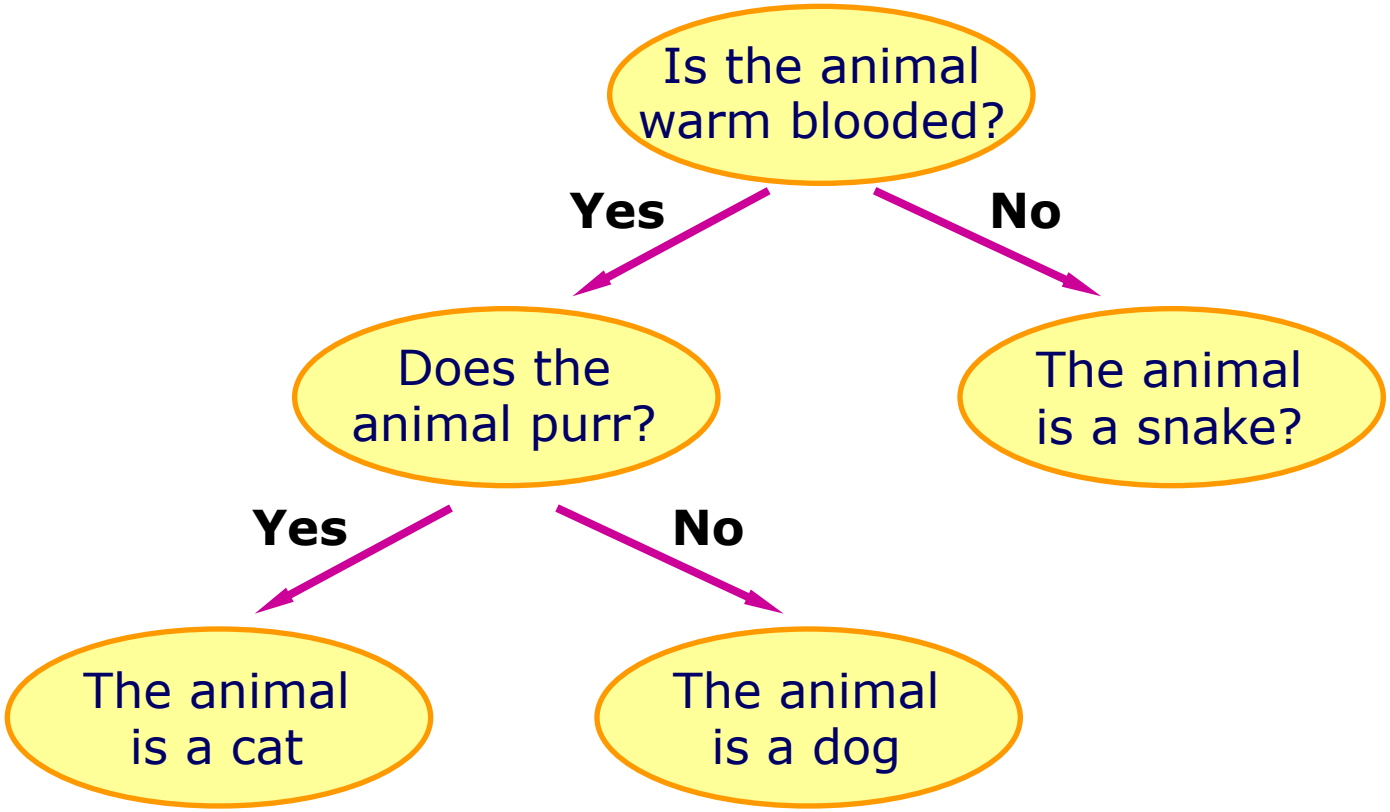
(dependents <fact-index-or-address>)

(dependencies < fact-index-or-address >)

Advanced pattern matching

Name	Notation	Example
Variable	?<variable-name>	?name
Single-field wildcard	?	?
Multi-field wildcard	$?	$?before
Connective constraint	~, |, &	?color&brown|black
Predicate function	or, not, and, test, >, <, >=, <=	?age&:(> ?age 18)
Return value	=	=(mod 13 4)
conditional element	or, not, and, exists, forall, logical	(forall <first-CE> <remaining-CEs>+)

Decision tree



Rule-based decision tree program

```
(deftemplate node
  (slot name)
  (slot type)
  (slot question)
  (slot yes-node)
  (slot no-node)
  (slot answer))
```

```
(defrule initialize
  (not (node (name root)))
  =>
  (load-facts "animal.dat")
  (assert (current-node root)))
```

Rule-based decision tree program

```
(node (name root) (type decision)
      (question "Is the animal warm-blooded?")
      (yes-node node1) (no-node node2) (answer nil))

(node (name node1) (type decision)
      (question "Does the animal purr?")
      (yes-node node3) (no-node node4) (answer nil))

(node (name node2) (type answer) (question nil)
      (yes-node nil) (no-node nil) (answer snake))

(node (name node3) (type answer) (question nil)
      (yes-node nil) (no-node nil) (answer cat))

(node (name node4) (type answer) (question nil)
      (yes-node nil) (no-node nil) (answer dog))
```

Rule-based decision tree program

```
(defrule ask-decision-node-question
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (question ?question))
   (not (answer ?))
   =>
   (printout t ?question " (yes or no) ")
   (assert (answer (read))))
```

```
(defrule bad-answer
  ?answer <- (answer ~yes&~no)
  =>
  (retract ?answer))
```

Rule-based decision tree program

```
(defrule proceed-to-yes-branch
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (yes-node ?yes-branch))
   ?answer <- (answer yes)
   =>
   (retract ?node ?answer)
   (assert (current-node ?yes-branch)))
```

Rule-based decision tree program

```
(defrule proceed-to-no-branch
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (no-node ?no-branch))
   ?answer <- (answer no)
   =>
   (retract ?node ?answer)
   (assert (current-node ?no-branch)))
```

Rule-based decision tree program

```
(defrule ask-if-answer-node-is-correct
   ?node <- (current-node ?name)
   (node (name ?name) (type answer) (answer ?value))
   (not (answer ?))
   =>
   (printout t "I guess it is a " ?value crlf)
   (printout t "Am I correct? (yes or no) ")
   (assert (answer (read))))
```

Rule-based decision tree program

```
(defrule answer-node-guess-is-correct
   ?node <- (current-node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer yes)
   =>
   (assert (ask-try-again))
   (retract ?node ?answer))
```

```
(defrule answer-node-guess-is-incorrect
   ?node <- (current-node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer no)
   =>
   (assert (replace-answer-node ?name))
   (retract ?answer ?node))
```

Rule-based decision tree program

```
(defrule ask-try-again
   (ask-try-again)
   (not (answer ?))
   =>
   (printout t "Try again? (yes or no) ")
   (assert (answer (read))))
```

Remember:

```
   (defrule bad-answer
     ?answer <- (answer ~yes&~no)
     =>
     (retract ?answer))
```

Rule-based decision tree program

```
(defrule one-more-time
   ?phase <- (ask-try-again)
   ?answer <- (answer yes)
   =>
   (retract ?phase ?answer)
   (assert (current-node root)))
```

```
(defrule no-more
  ?phase <- (ask-try-again)
  ?answer <- (answer no)
  =>
  (retract ?phase ?answer)
  (save-facts "animal.dat" local node))
```

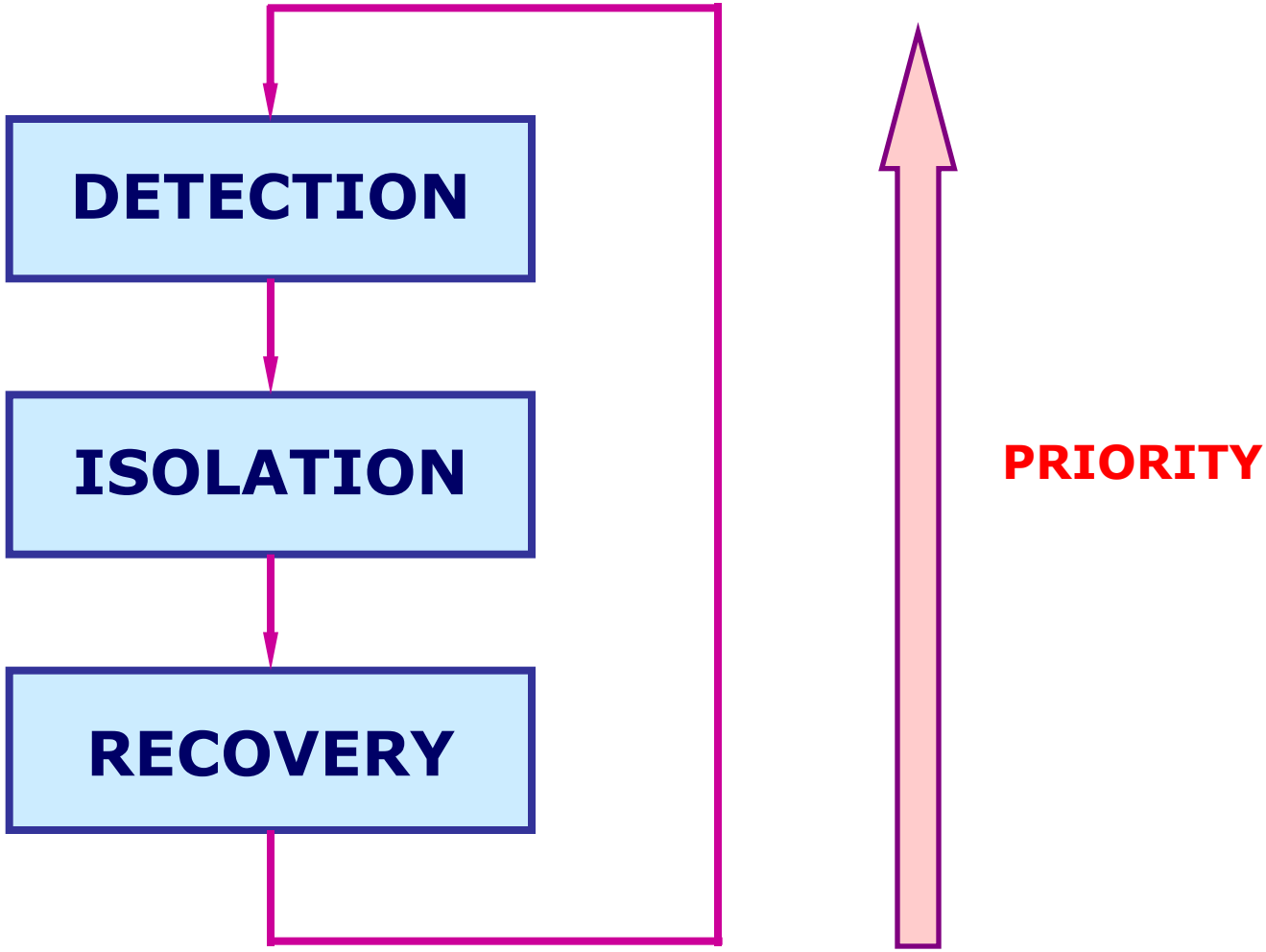
Rule-based decision tree program

```
(defrule replace-answer-node
   ?phase <- (replace-answer-node ?name)
   ?data <- (node (name ?name)
                  (type answer)
                  (answer ?value))
   =>
   (retract ?phase)
   ; Determine what the guess should have been
   (printout t "What is the animal? ")
   (bind ?new-animal (read))
   ; Get the question for the guess
   (printout t "What question when answered yes ")
   (printout t "will distinguish " crlf "   a ")
   (printout t ?new-animal " from a " ?value "? ")
   (bind ?question (readline))
   (printout t "Now I can guess " ?new-animal crlf)
```

Rule-based decision tree program

```
; Create the new learned nodes
   (bind ?newnode1 (gensym*))
   (bind ?newnode2 (gensym*))
   (modify ?data (type decision)
                 (question ?question)
                 (yes-node ?newnode1)
                 (no-node ?newnode2))
   (assert (node (name ?newnode1)
                 (type answer)
                 (answer ?new-animal)))
   (assert (node (name ?newnode2)
                 (type answer)
                 (answer ?value)))
   ; Determine if the player wants to try again
   (assert (ask-try-again)))
```

Phases



Salience

Purpose

Specify the priority of rules

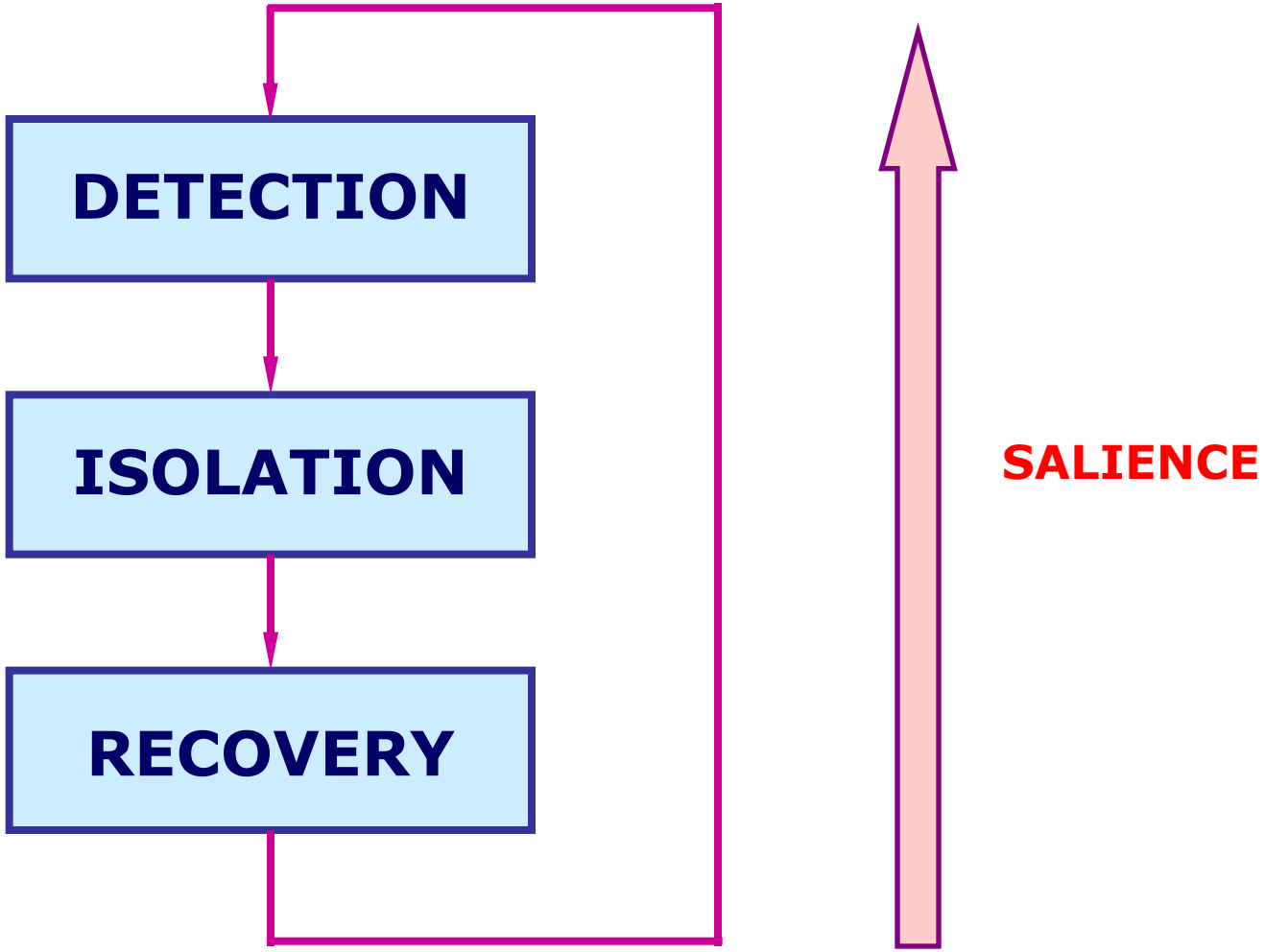
Syntax

(declare (salience <integer>)

Range

(–10,000 10,000), 0 (default)

Salience approach

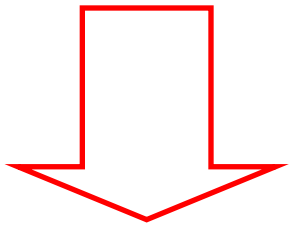


Misuse of salience

```
(defrule situation-emergency
  (declare (salience 10))
  (situation emergency)
  =>
  (assert (action emergency)))

(defrule situation-important
  (declare (salience 5))
  (situation important)
  =>
  (assert (action important)))

(defrule situation-normal
  (declare (salience 0))
  (situation normal)
  =>
  (assert (action normal)))
```



Misuse of salience

```
(defrule situation-emergency
  (situation emergency)
  =>
  (assert (action emergency)))

(defrule situation-important
  (situation important)
  (not (situation emergency))
  =>
  (assert (action important)))
```

```
(defrule situation-normal
  (situation normal)
  (not (situation emergency))
  (not (situation important))
  =>
  (assert (action normal)))
```

The tight interaction between the rules is removed.

Control pattern

```
(defrule detection-rule
  (phase detection)
  <patterns>*
  =>
  <action>* )

(defrule isolation-rule
  (phase isolation)
  <patterns>*
  =>
  <action>* )

(defrule recovery-rule
  (phase recovery)
  <patterns>*
  =>
  <action>* )
```

Control pattern

```
(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)))

(defrule find-fault-location-and-recovery
  (phase recovery)
  <pattern>* => <actions>*)
```

Lower than default

defmodule

Syntax

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

Default module

MAIN::

Command

(get-current-module)
(set-current-module)
(focus <module-name>+)
(list-focus-stack)
(clear-focus-stack)
(pop-focus)
(get-focus)

defmodule approach

```
(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-phase)
  =>
  (focus ?next-phase)
  (retract ?list)
  (assert (phase-sequence ?other-phases ?next-phase))))
```

Import and export facts

```
(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))
```

Syntax

(export deftemplate <deftemplate-name>+)
(import deftemplate <deftemplate-name>+)

initial-fact

Special rules

```
(defrule <rule-name> [<optional-comment>]
    (not (<pattern>))
    <patterns>*
    =>
    <actions>*

(defrule <rule-name> [<optional-comment>]
    =>
    <actions>*
```

Export

```
(defmodule MAIN
  (export deftemplate initial-fact))
```

Import

```
(defmodule <module-name>
  (import MAIN deftemplate initial-fact))
```

Procedural functions

If function

```
(if <predicate-expression>
  then <expression>+
  [else <expression>+])
```

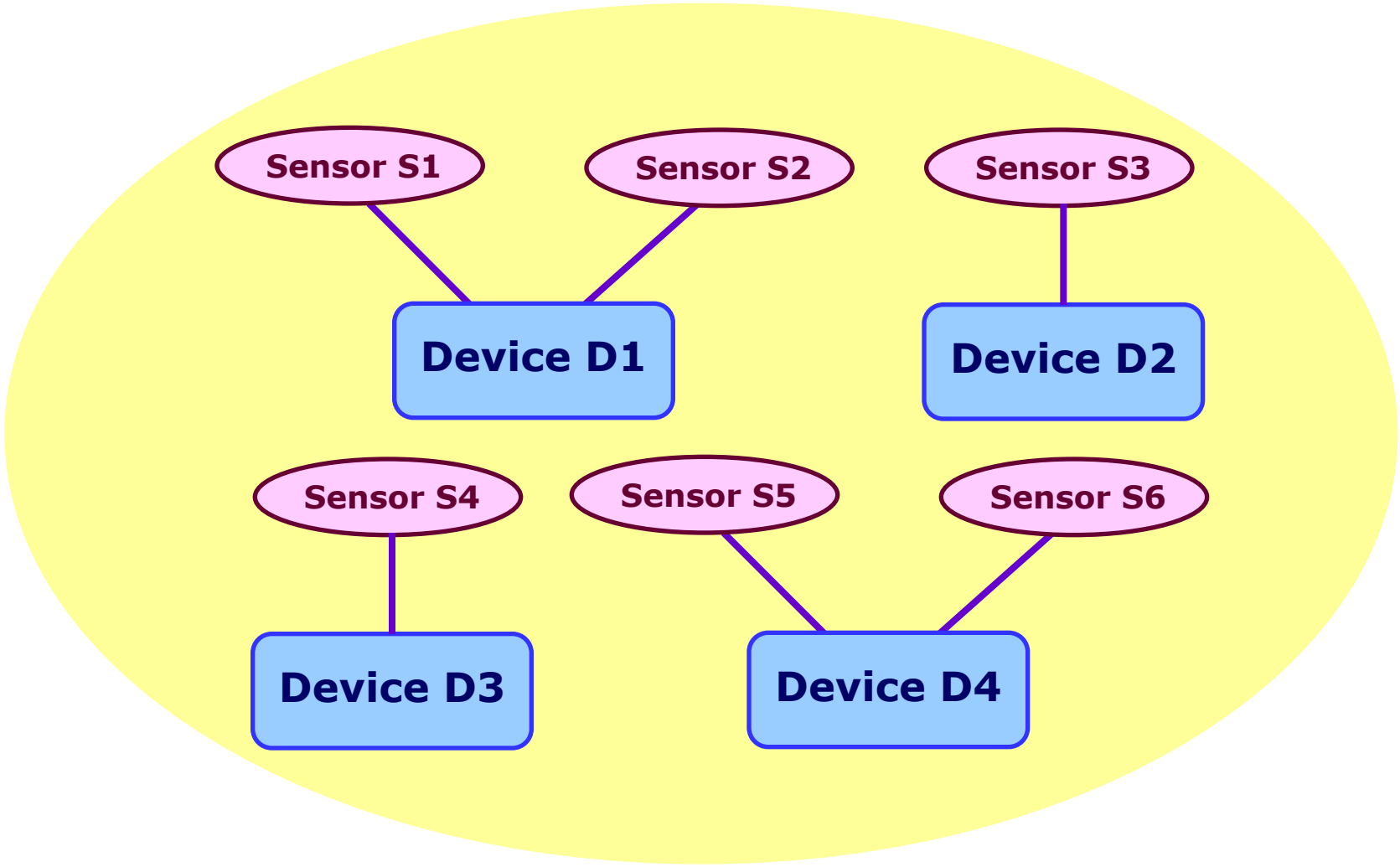
While function

```
(while <predicate-expression> [do]
  <expression>+)
```

Example

```
(defrule continue-check
  ?phase <- (phase check-continue)
  =>
  (retract ?phase)
  (printout t "Continue? ")
  (bind ?answer (read))
  (while (and (neq ?answer yes) (neq ?answer no))
    do (printout t "Continue? ")
       (bind ?answer (read)))
  (if (eq ?answer yes)
    then (assert (phase continue))
    else  (halt))))
```

Monitoring problem



Sensor Attributes

Sensor	Low Red Line	Low Guard Line	High Guard line	High Red Line
S1	60	70	120	130
S2	20	40	160	180
S3	60	70	120	130
S4	60	70	120	130
S5	65	70	120	125
S6	110	115	125	130

If less than or equal to low red line
or greater than or equal to high red line
then shut down device

If great than low red line and less than or equal to low guard line
or great than or equal to high guard line and less than high red line
then issue warning or shut down device after given cycles

Monitoring problem

```
(defmodule MAIN (export ?ALL))

(deftemplate MAIN::device
   (slot name (type SYMBOL))
   (slot status (allowed-values on off)))

(deffacts MAIN::device-information
  (device (name D1) (status on))
  (device (name D2) (status on))
  (device (name D3) (status on))
  (device (name D4) (status on)))
```

Monitoring problem

```
(deftemplate MAIN::sensor
   (slot name (type SYMBOL))
   (slot device (type SYMBOL))
   (slot raw-value (type SYMBOL NUMBER)
                   (allowed-symbols none)
                   (default none))
   (slot state (allowed-values low-red-line
                               low-guard-line
                               normal
                               high-red-line
                               high-guard-line)
               (default normal))
   (slot low-red-line (type NUMBER))
   (slot low-guard-line (type NUMBER))
   (slot high-guard-line (type NUMBER))
   (slot high-red-line (type NUMBER)))
```

Monitoring problem

```
(deffacts MAIN::sensor-information
  (sensor (name S1)
          (device D1)
          (low-red-line 60)
          (low-guard-line 70)
          (high-guard-line 120)
          (high-red-line 130))
  (sensor (name S2)
          (device D1)
          (low-red-line 20)
          (low-guard-line 40)
          (high-guard-line 160)
          (high-red-line 180))
  ...)
```

Monitoring problem

```
(deffacts MAIN::cycle-start
  (data-source facts)
  (cycle 0))

(defrule MAIN::Begin-Next-Cycle
   ?f <- (cycle ?current-cycle)
   =>
   (retract ?f)
   (assert (cycle (+ ?current-cycle 1)))
   (focus INPUT TRENDS WARNINGS))
```

Monitoring problem

```
(defmodule INPUT (import MAIN ?ALL))

(deftemplate INPUT::fact-data-for-sensor
   (slot name)
   (multislot data))

(deffacts INPUT::sensor-fact-data-values
  (fact-data-for-sensor (name S1)
     (data 100 100 110 110 115 120))
  (fact-data-for-sensor (name S2)
     (data 110 120 125 130 130 135))
  ...)
```

Monitoring problem

```
(defrule INPUT::Read-Sensor-Values-From-Facts
  (data-source facts)
  ?s <- (sensor (name ?name) (raw-value none))
  ?f <- (fact-data-for-sensor
              (name ?name) (data ?raw-value $?rest))
  =>
  (modify ?s (raw-value ?raw-value))
  (modify ?f (data ?rest)))

(defrule INPUT::No-Fact-Data-Values-Left
  (data-source facts)
  (sensor (name ?name) (raw-value none))
  (fact-data-for-sensor (name ?name) (data))
  =>
  (printout t "No fact data for sensor " ?name crlf)
  (printout t "Halting monitoring system" crlf)
  (halt))
```

Monitoring problem

```
(defmodule TRENDS (import MAIN ?ALL))

(defrule TRENDS::Normal-State
  ?s <- (sensor (raw-value ?raw-value&~none)
                (low-guard-line ?lgl)
                (high-guard-line ?hgl))
  (test (and (> ?raw-value ?lgl) (< ?raw-value ?hgl)))
  =>
  (modify ?s (state normal) (raw-value none)))
```

Monitoring problem

```
(defrule TRENDS::High-Guard-Line-State
  ?s <- (sensor (raw-value ?raw-value&~none)
                (high-guard-line ?hgl)
                (high-red-line ?hrl))
  (test (and (>= ?raw-value ?hgl) (< ?raw-value ?hrl)))
  =>
  (modify ?s (state high-guard-line) (raw-value none)))
```

Other rules:

High-Red-Line-State
Low-Guard-Line-State
Low-Red-Line-State

Monitoring problem

```
(deftemplate MAIN::sensor-trend
   (slot name)
   (slot state (default normal))
   (slot start (default 0))
   (slot end (default 0))
   (slot shutdown-duration (default 3)))

(deffacts MAIN::start-trends
  (sensor-trend (name S1) (shutdown-duration 3))
  (sensor-trend (name S2) (shutdown-duration 5))
  (sensor-trend (name S3) (shutdown-duration 4))
  (sensor-trend (name S4) (shutdown-duration 4))
  (sensor-trend (name S5) (shutdown-duration 4))
  (sensor-trend (name S6) (shutdown-duration 2)))
```

Monitoring problem

```
(defrule TRENDS::State-Has-Not-Changed
  (cycle ?time)
  ?trend <- (sensor-trend (name ?sensor)
                          (state ?state)
                          (end ?end-cycle&~?time))
  (sensor (name ?sensor)
          (state ?state)
          (raw-value none))
  =>
  (modify ?trend (end ?time)))
```

Monitoring problem

```
(defrule TRENDS::State-Has-Changed
  (cycle ?time)
  ?trend <- (sensor-trend (name ?sensor)
                          (state ?state)
                          (end ?end-cycle&~?time))
  (sensor (name ?sensor)
          (state ?new-state&~?state)
          (raw-value none))
  =>
  (modify ?trend (start ?time)
                 (end ?time)
                 (state ?new-state)))
```

Monitoring problem

```
(defmodule WARNINGS (import MAIN ?ALL))

(defrule WARNINGS::Shutdown-In-Red-Region
  (cycle ?time)
  (sensor-trend
     (name ?sensor)
     (state ?state&high-red-line | low-red-line))
  (sensor (name ?sensor) (device ?device))
  ?on <- (device (name ?device) (status on))
  =>
  (printout t "Cycle " ?time " - ")
  (printout t "Sensor " ?sensor " in " ?state crlf)
  (printout t "   Shutting down device " ?device crlf)
  (modify ?on (status off)))
```

Monitoring problem

```
(defrule WARNINGS::Shutdown-In-Guard-Region
  (cycle ?time)
  (sensor-trend
      (name ?sensor)
      (state ?state&high-guard-line | low-guard-line)
      (shutdown-duration ?length)
      (start ?start) (end ?end))
  (test (>= (+ (- ?end ?start) 1)  ?length))
  (sensor (name ?sensor) (device ?device))
  ?on <- (device (name ?device) (status on))
  =>
  (printout t "Cycle " ?time " - ")
  (printout t "Sensor " ?sensor " in " ?state " ")
  (printout t "for " ?length " cycles " crlf)
  (printout t "   Shutting down device " ?device crlf)
  (modify ?on (status off)))
```

Monitoring problem

```
(defrule WARNINGS::Sensor-In-Guard-Region
  (cycle ?time)
  (sensor-trend
      (name ?sensor)
      (state ?state&high-guard-line | low-guard-line)
      (shutdown-duration ?length)
      (start ?start)
      (end ?end))
  (test (< (+ (– ?end ?start) 1) ?length))
  =>
  (printout t "Cycle " ?time " - ")
  (printout t "Sensor " ?sensor " in " ?state crlf))
```

Inference engine

while not done

Conflict Resolution: If there are activations, then select one with the highest priority else done.

Act: Sequentially perform the actions on the RHS of the selected activation. Remove the activation that has just fired form the agenda.

Match: Update the agenda by checking whether the LHSs of any rules are satisfied. If so, activate them. Remove activations if the LHSs of their rules are no longer satisfied.

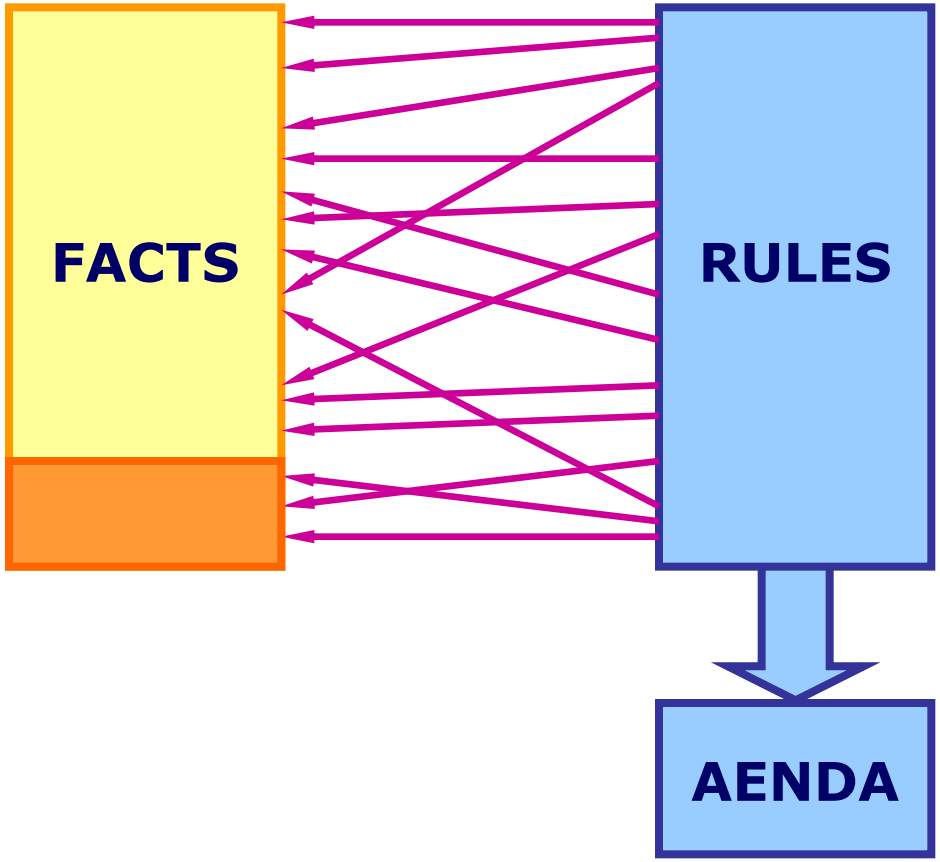
Check for Halt: If a halt action is performed or a break command given, then done.

end-while

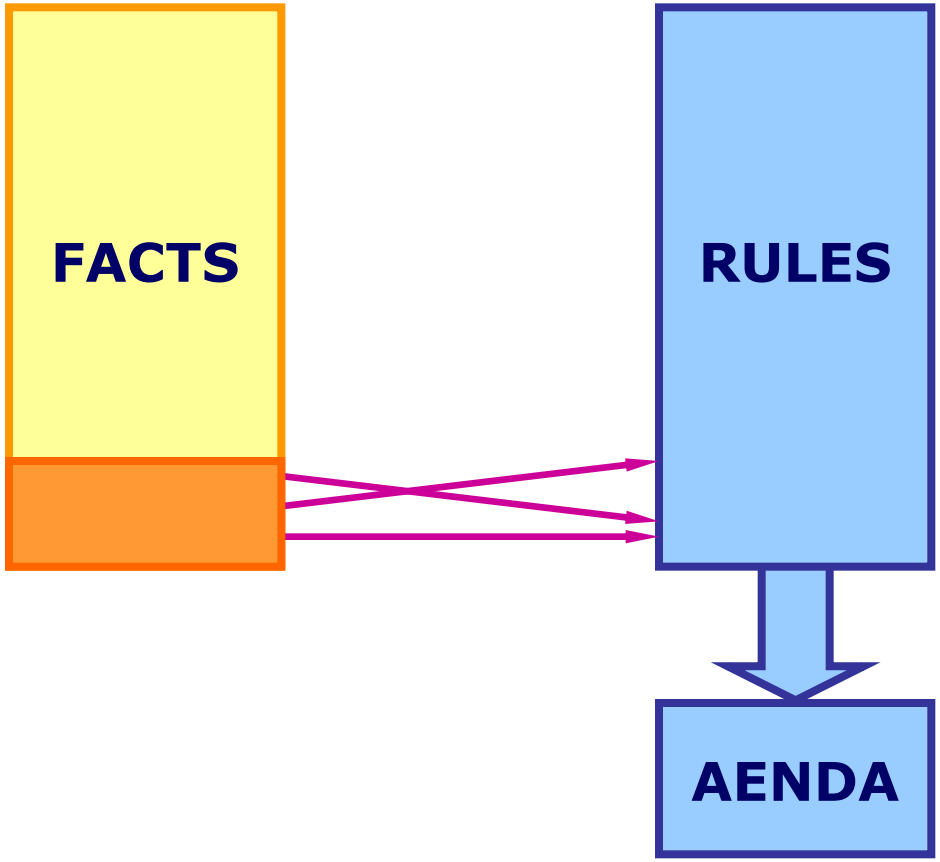
Accept a new user command

Recognize-act cycle

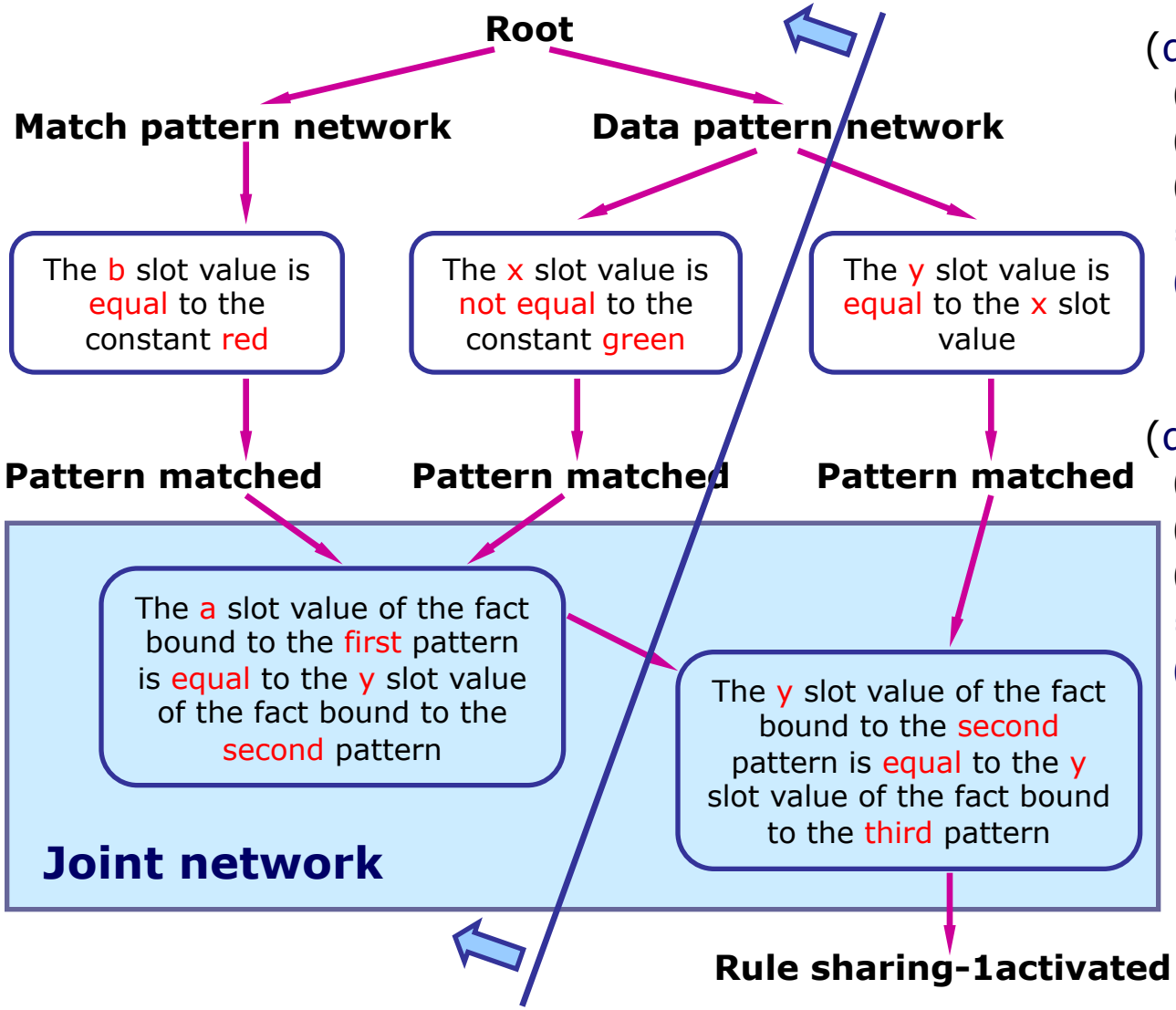
Rules searching for facts



Rules searching for facts



Pattern network




```
(defrule sharing-1
  (match (a ?x) (b red))
  (data (x ~green) (y ?x))
  (data (x ?x) (y ?x))
  =>
  (printout t "Matching
    sharing-1 rule. " crlf ))
```

```
(defrule sharing-2
  (match (a ?y) (b red))
  (data (x ~green) (y ?y))
  (data (x ?y) (y ?z))
  =>
  (printout t "Matching
    sharing-2 rule. " crlf ))
```

Efficiency

- Most specific patterns go first.

Importance of pattern order

```
(defrule good-match
  (find-match ?x ?y ?z ?w)   Pattern 1
  (item ?x)                  Pattern 2
  (item ?y)                  Pattern 3
  (item ?z)                  Pattern 4
  (item ?w)                  Pattern 5
  =>
  (assert (found-match ?x ?y ?z ?w)))
```

```
(deffacts information)
  (find-match a c e g)   f-1
  (item a)               f-2
  (item b)               f-3
  (item c)               f-4
  (item d)               f-5
  (item e)               f-6
  (item f)               f-7
  (item g))              f-8
```

Rule good-match has the following 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

Importance of pattern order

```
(defrule good-match
  (find-match ?x ?y ?z ?w)   Pattern 1
  (item ?x)                  Pattern 2
  (item ?y)                  Pattern 3
  (item ?z)                  Pattern 4
  (item ?w)                  Pattern 5
  =>
  (assert (found-match ?x ?y ?z ?w)))
```

```
(deffacts information)
  (find-match a c e g)   f-1
  (item a)               f-2
  (item b)               f-3
  (item c)               f-4
  (item d)               f-5
  (item e)               f-6
  (item f)               f-7
  (item g))              f-8
```

Rule good-match has the following partial matches:

Pattern 1: [f-1]
Pattern 1-2: [f-1, f-2]
Pattern 1-3: [f-1, f-2, f-4]
Pattern 1-4: [f-1, f-2, f-4, f-6]
Pattern 1-5: [f-1, f-2, f-4, f-6, f-8]

Importance of pattern order

```
(defrule bad-match
  (item ?x)                          Pattern 1
  (item ?y)                          Pattern 2
  (item ?z)                          Pattern 3
  (item ?w)                          Pattern 4
  (find-match ?x ?y ?z ?w)           Pattern 5
  =>
  (assert (found-match ?x ?y ?z ?w)))
```

```
(deffacts information)
  (find-match a c e g)   f-1
  (item a)               f-2
  (item b)               f-3
  (item c)               f-4
  (item d)               f-5
  (item e)               f-6
  (item f)               f-7
  (item g))              f-8
```

Rule bad-match has the following 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

Importance of pattern order

Rule bad-match has the following partial matches:

Pattern 1: [f-2, f-3, f-4, f-5, f-6, f-7, f-8]

Pattern 1-2: [f-2, f-2], [f-2, f-3], [f-2, f-4], [f-2, f-5],
[f-2, f-6], [f-2, f-7], [f-2, f-8],
[f-3, f-2], [f-3, f-3], [f-3, f-4], [f-3, f-5],
[f-3, f-6], [f-3, f-7], [f-3, f-8],
[f-4, f-2], [f-4, f-3], [f-4, f-4], [f-4, f-5],
[f-4, f-6], [f-4, f-7], [f-4, f-8],
[f-5, f-2], [f-5, f-3], [f-5, f-4], [f-5, f-5],
[f-5, f-6], [f-5, f-7], [f-5, f-8],
[f-6, f-2], [f-6, f-3], [f-6, f-4], [f-6, f-5],
[f-6, f-6], [f-6, f-7], [f-6, f-8],
[f-7, f-2], [f-7, f-3], [f-7, f-4], [f-7, f-5],
[f-7, f-6], [f-7, f-7], [f-7, f-8],
[f-8, f-2], [f-8, f-3], [f-8, f-4], [f-8, f-5],
[f-8, f-6], [f-8, f-7], [f-8, f-8],

…

Bad rule may run out of the memory

Efficiency

- Most specific patterns go first.
- Patterns matching volatile facts go last.
- Patterns matching the fewest facts go first.
- Limit the number of multifield wildcards and variables.

Multifield wildcards

Pattern: (list (items $?a $?b $?c)) Fact: (list (items x 5 y 9))

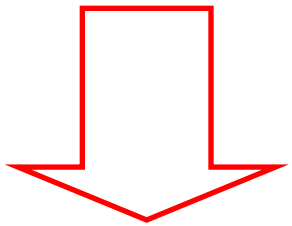
Match attempt	Fields matched by $?a	Fields matched by $?b	Fields matched by $?c
1			x5y9
2		x	5y9
3		x5	y9
4		x5y	9
5		x5y9	
6	x		5y9
7	x	5	y9
8	x	5y	9
9	x	5y9	
10	x5		y9
11	x5	y	9
12	x5	y9	
13	x5y		9
14	x5y	9	
15	x5y9		

Efficiency

- Most specific patterns go first.
- Patterns matching volatile facts go last.
- Patterns matching the fewest facts go first.
- Limit the number of multifield wildcards and variables.
- Test conditional elements should be placed as close to the top of the rule as possible.

Test conditional elements

```
(defrule three-distinct-points
  ?point-1 <- (point (x ?x1) (y ?y1))
  ?point-2 <- (point (x ?x2) (y ?y2))
  ?point-3 <- (point (x ?x3) (y ?y3))
  (test (and (neq ?point-1 ?point-2)
             (neq ?point-2 ?point-3)
             (neq ?point-1 ?point-3)))
  =>
  (assert (distinct-points (x1 ?x1) (y1 ?y1)
                           (x2 ?x2) (y2 ?y2)
                           (x3 ?x3) (y3 ?y3))))
```



Test conditional elements

```
(defrule three-distinct-points
  ?point-1 <- (point (x ?x1) (y ?y1))
  ?point-2 <- (point (x ?x2) (y ?y2))
  (test (neq ?point-1 ?point-2))
  ?point-3 <- (point (x ?x3) (y ?y3))
  (test (and (neq ?point-2 ?point-3)
             (neq ?point-1 ?point-3)))
  =>
  (assert (distinct-points (x1 ?x1) (y1 ?y1)
                           (x2 ?x2) (y2 ?y2)
                           (x3 ?x3) (y3 ?y3))))
```

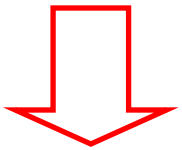
Efficiency

- Most specific patterns go first.
- Patterns matching volatile facts go last.
- Patterns matching the fewest facts go first.
- Limit the number of multifield wildcards and variables.
- Test conditional elements should be placed as close to the top of the rule as possible.
- Use built-in pattern-matching (connective) constraints instead of other equivalent expressions.

Build-in pattern-matching constraints

```
(defrule primary-color
  (color ?x&:(or (eq ?x red)
                 (eq ?x green)
                 (eq ?x blue)))
  =>
  (assert (primary-color ?x)))
```

Two patterns



```
(defrule primary-color
  (color ?x&red | green | blue)
  =>
  (assert (primary-color ?x)))
```

Efficiency

- Most specific patterns go first.
- Patterns matching volatile facts go last.
- Patterns matching the fewest facts go first.
- Limit the number of multifield wildcards and variables.
- Test conditional elements should be placed as close to the top of the rule as possible.
- Use built-in pattern-matching (connective) constraints instead of other equivalent expressions.
- reduce the number of facts and load facts only when they are needed.

Loading and saving facts

Syntax

(load-facts <file-name>)

(save-facts <file-name> [<save-scope> <deftemplate-names>*])
where <save-scope> is defined as visible | local

Example for facts file format

(data 34)
(data 89)
...

Command example

(load-facts "facts.dat")

Any question?

Xiaoqing Zheng
Fudan University