Jess Language: Pattern Matching

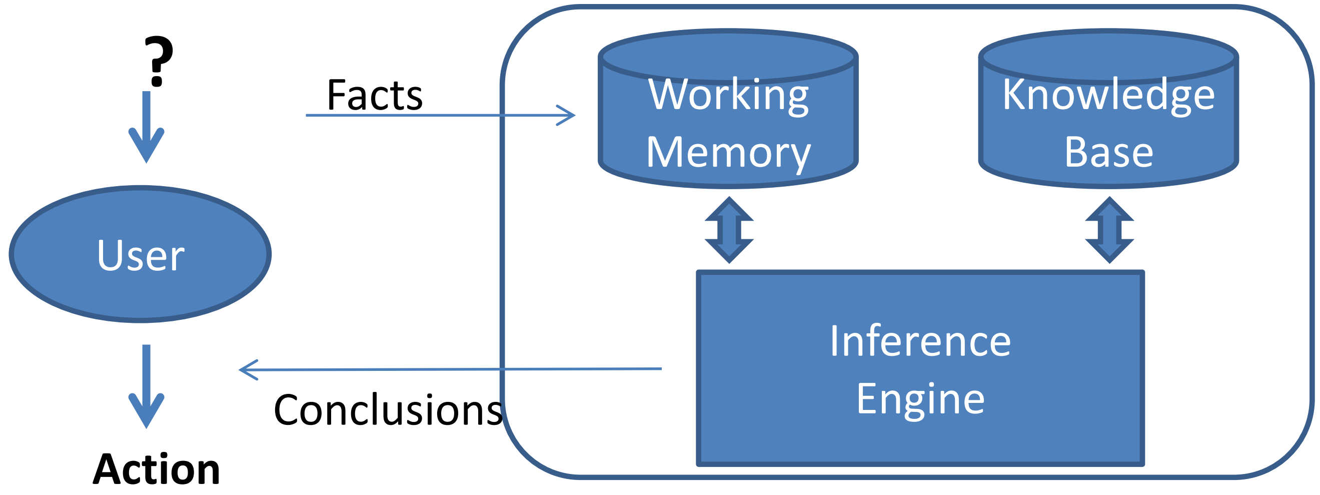
The Jess Language Part 2
CS3025, Knowledge-Based Systems
Lecture 09

Yuting Zhao
Yuting.zhao@gmail.com

2017-10-24

Warm Up: what we have learnt?

- Knowledge-based system:
  - Inference Engine, Facts, Rules



  - rule
  - Variables

```
(defrule mortality
       (is-a ?person man)
       =>
       (assert (is-a ?person mortal))
```

Outline

- Pattern matching
  1. Exact match
  2. Match variables
  3. Unordered/deftemplate Patterns
  4. Multifield Variables
  5. Wildcards
- Conditional Elements
  1. and, or, not
  2. test
  3. exists
  4. Constraints
- Manipulating Lists

Pattern Matching: (1) **Exact Match**

- When checking if a rule is activated?
- **Pattern matching**:

  Is there a match between the pattern in the LHS and facts in WM?

```
(defrule mortality
?       (is-a Socrates man)
        =>
        (assert (is-a Socrates mortal))
```

fact: (is-a Plato man)

fact: (is-a Socrates man)

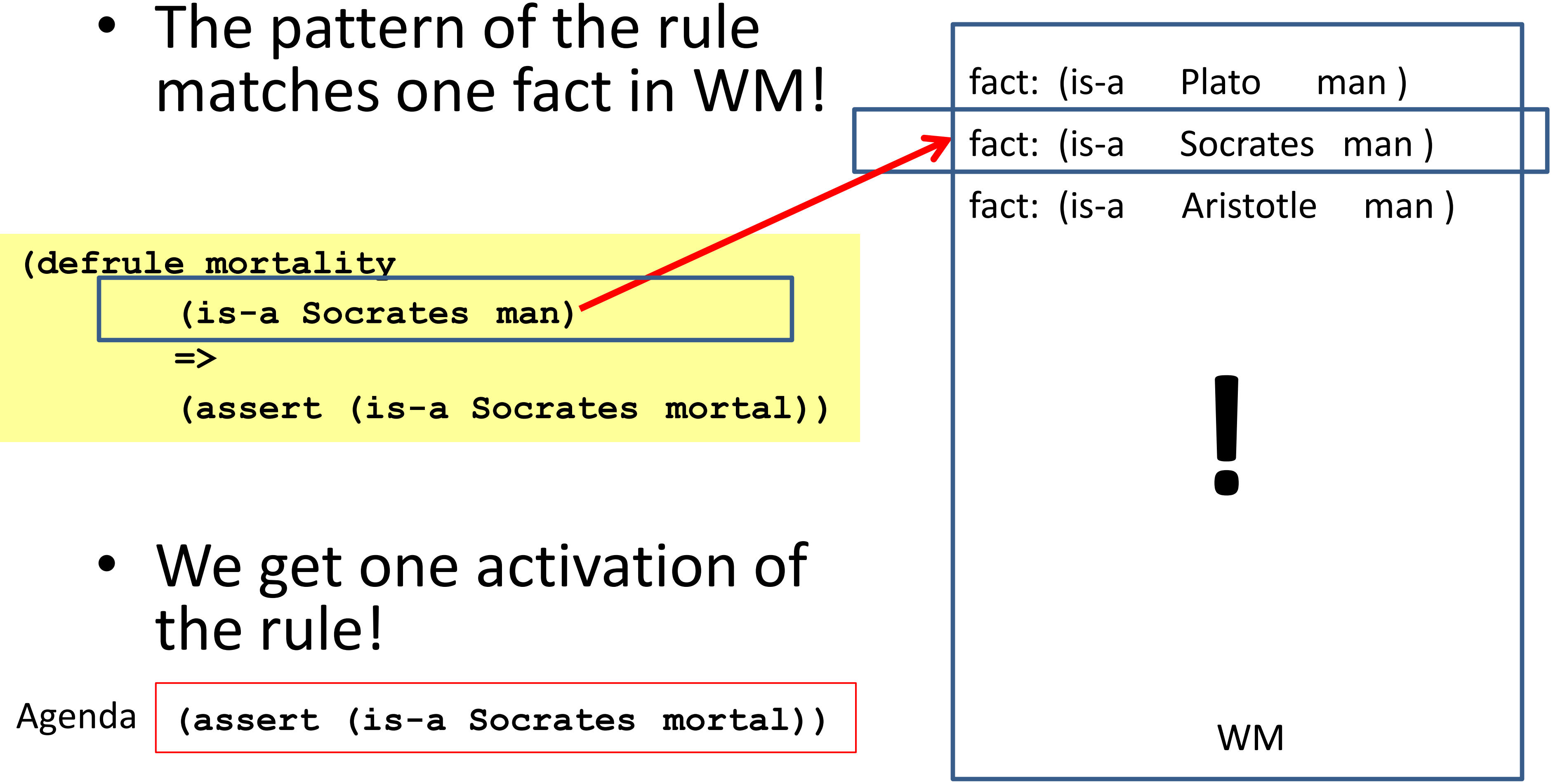
fact: (is-a Aristotle man)

WM

Rule Activation

Exact Match, **Rule Activation**

- The pattern of the rule matches one fact in WM!



```
(defrule mortality
         (is-a Socrates man)
         =>
         (assert (is-a Socrates mortal))
```

fact: (is-a Plato man)

fact: (is-a Socrates man)

fact: (is-a Aristotle man)

!

WM

- We get one activation of the rule!

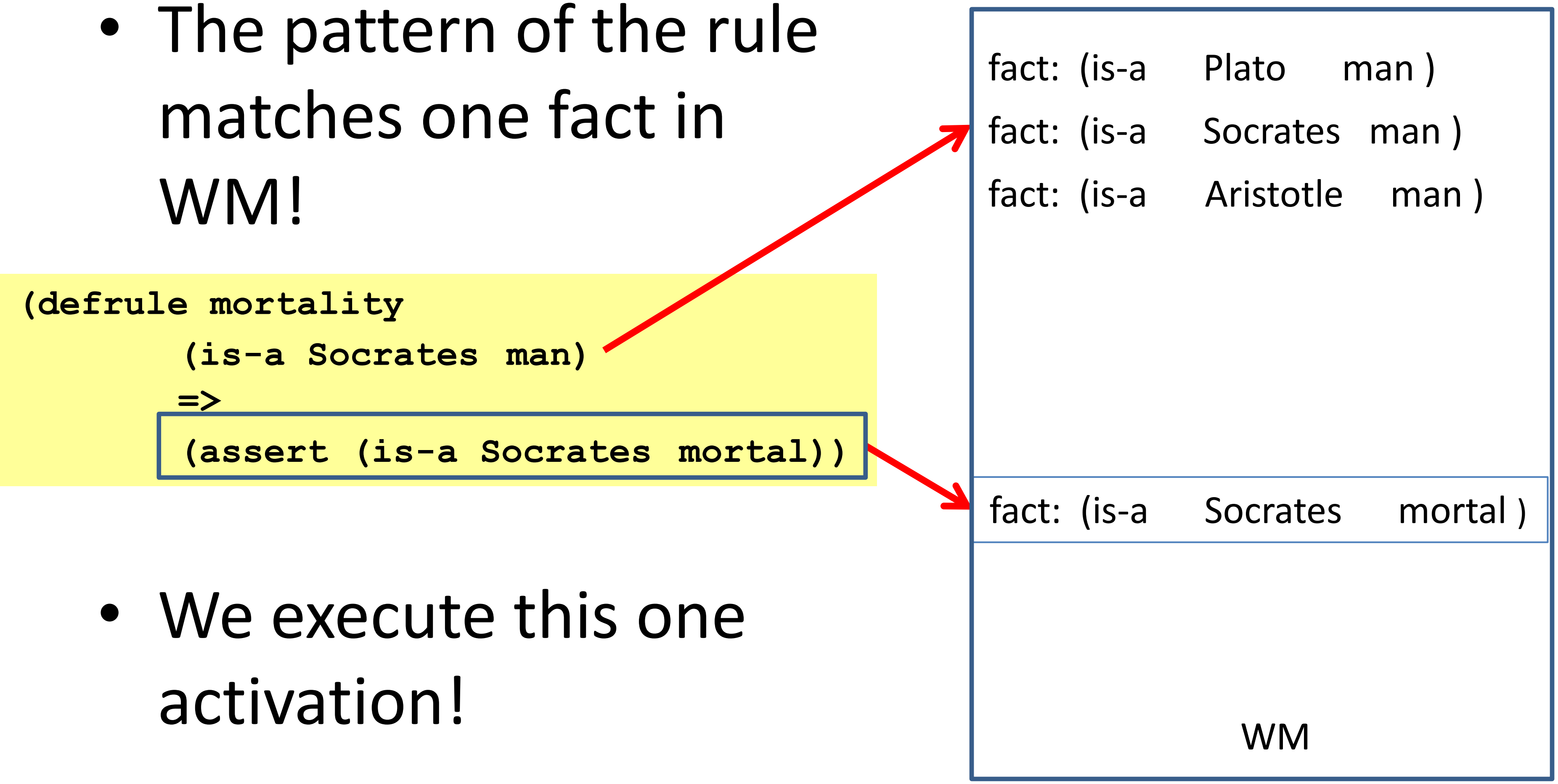
Agenda

```
(assert (is-a Socrates mortal))
```

Rule Execution

Exact Match

- The pattern of the rule matches one fact in WM!



```
(defrule mortality
       (is-a Socrates man)
      =>
       (assert (is-a Socrates mortal))
```

fact: (is-a Plato man)

fact: (is-a Socrates man)

fact: (is-a Aristotle man)

fact: (is-a Socrates mortal)

- We execute this one activation!

Pattern Matching: (2) Match variables

- When we use **variables** in rules, is there a match between the pattern and facts in WM?

```
(defrule mortality
          (is-a ?person man)
          =>
          (assert (is-a ?person mortal))
```

?

fact: (is-a Plato man)

fact: (is-a Socrates man)

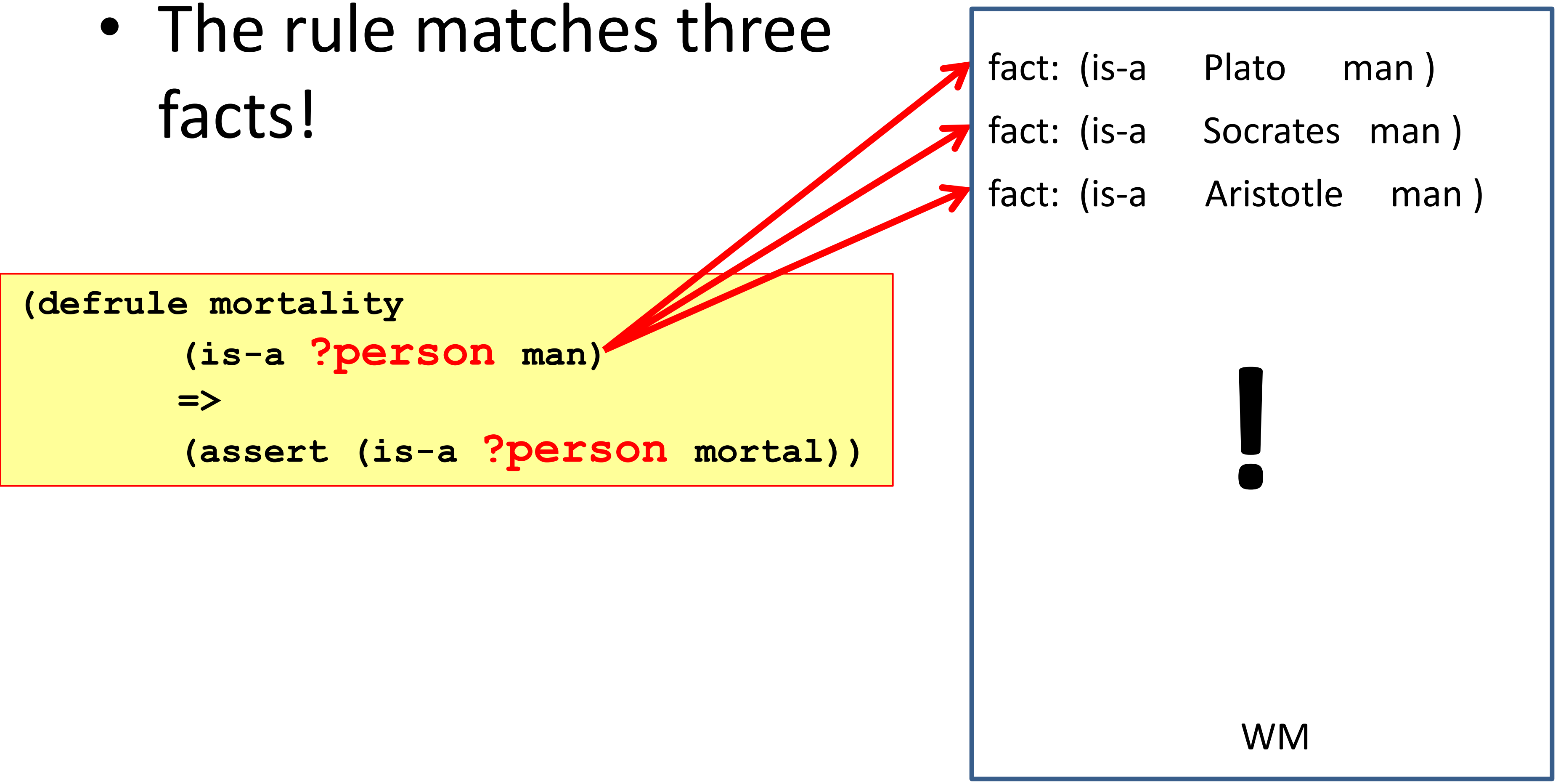
fact: (is-a Aristotle man)

WM

Pattern Matching

Match of Sets of Facts

- The rule matches three facts!



```
(defrule mortality
        (is-a ?person man)
        =>
        (assert (is-a ?person mortal))
```

fact: (is-a Plato man)

fact: (is-a Socrates man)

fact: (is-a Aristotle man)

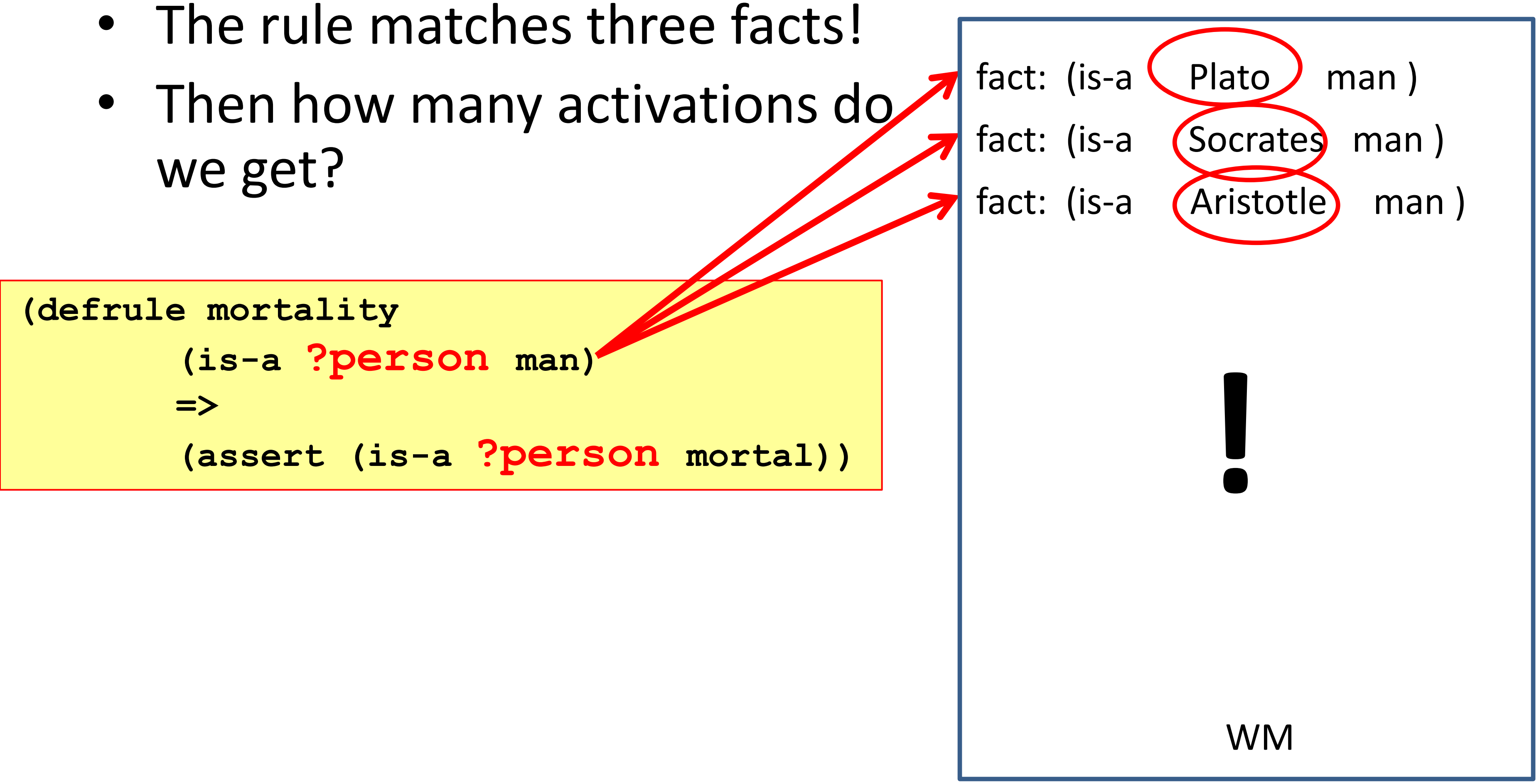
!

WM

Multiple Activations of a Rule

Match of **Sets of Facts**

- The rule matches three facts!
- Then how many activations do we get?



```
(defrule mortality
        (is-a ?person man)
        =>
        (assert (is-a ?person mortal))
```

fact: (is-a Plato man)

fact: (is-a Socrates man)

fact: (is-a Aristotle man)

!

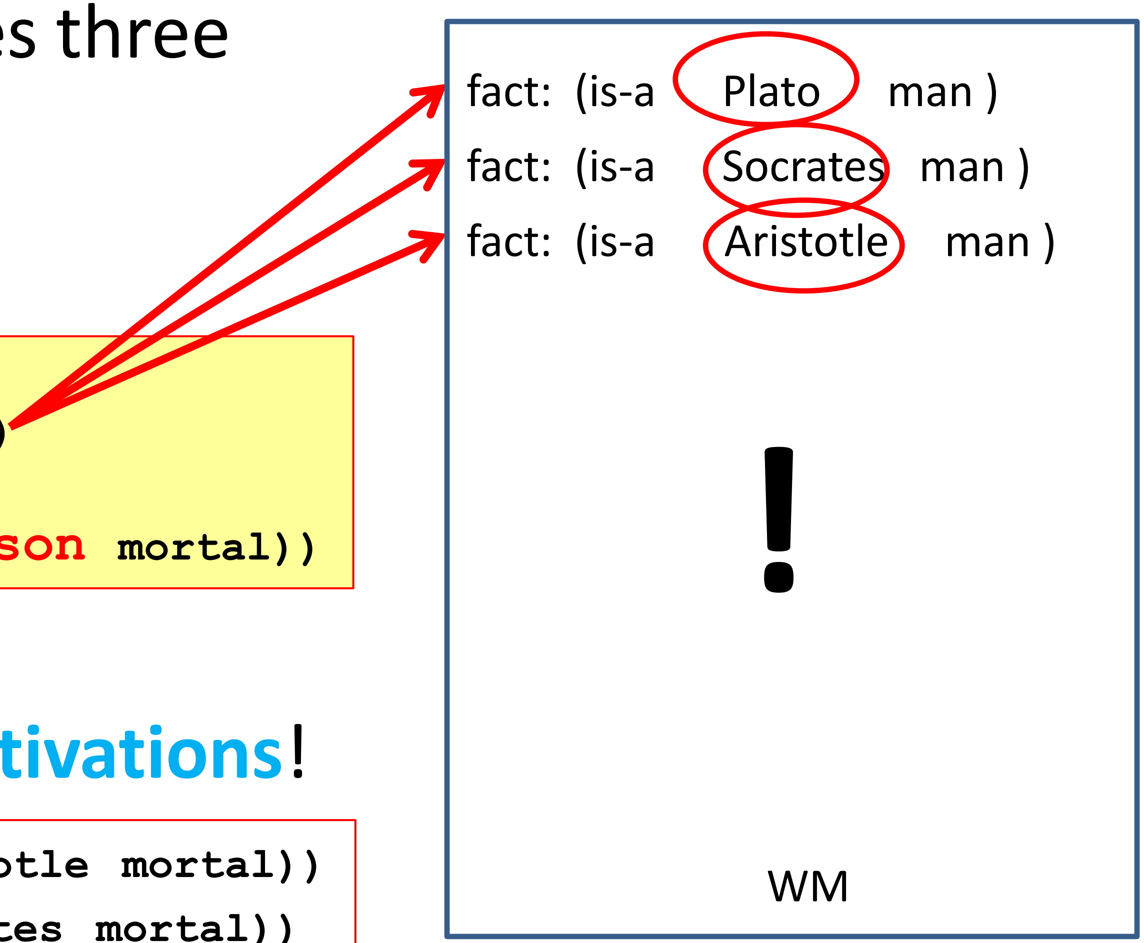
WM

Multiple Activations of a Rule

Match of Sets of Facts

- The rule matches three facts!

```
(defrule mortality
        (is-a ?person man)
        =>
        (assert (is-a ?person mortal))
```



fact: (is-a Plato man)

fact: (is-a Socrates man)

fact: (is-a Aristotle man)

!

WM

- We get **three activations**!

Agenda

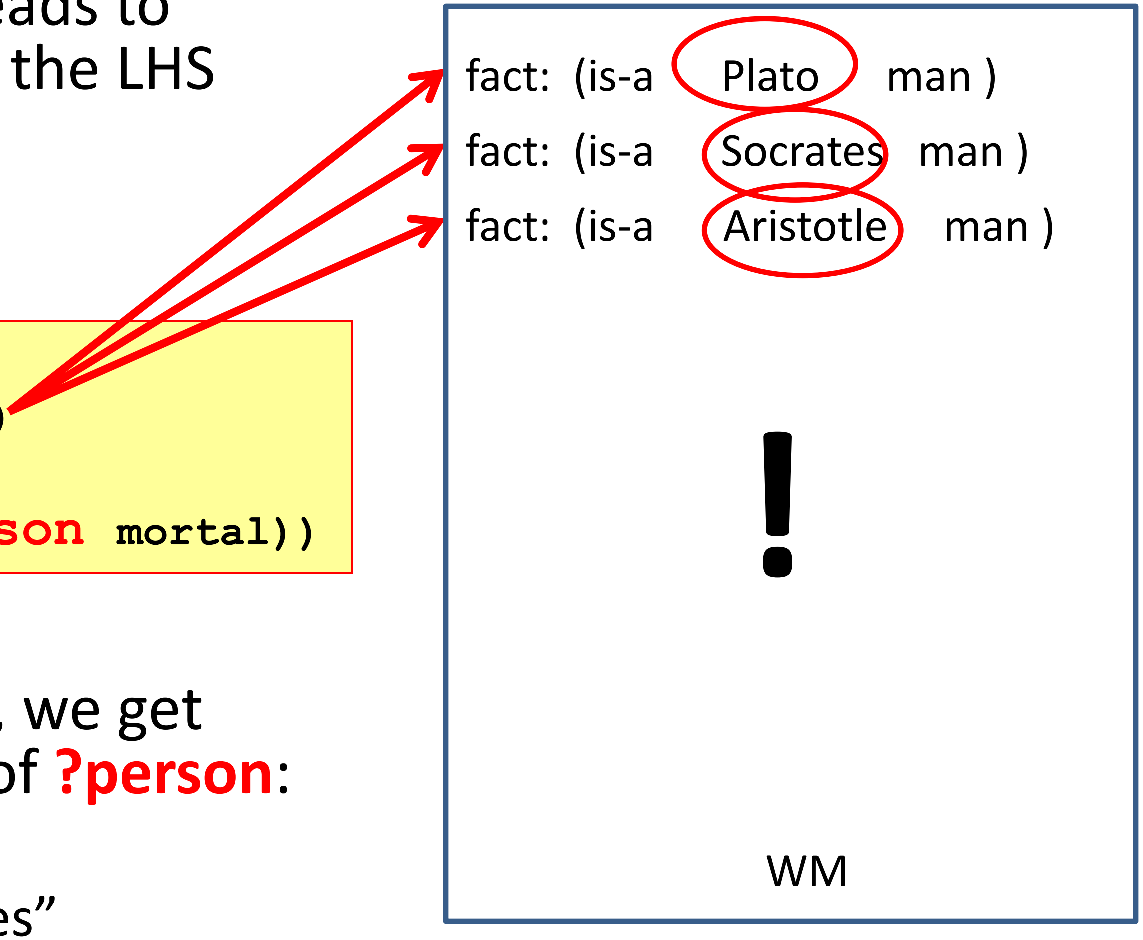
```
(assert (is-a Aristotle mortal))
(assert (is-a Socrates mortal))
(assert (is-a Plato mortal))
```

Pattern Matching – Variable Binding

Match Sets of Facts

- Pattern matching leads to variable binding on the LHS

```
(defrule mortality
         (is-a ?person man)
         =>
         (assert (is-a ?person mortal))
```



fact: (is-a Plato man)

fact: (is-a Socrates man)

fact: (is-a Aristotle man)

!

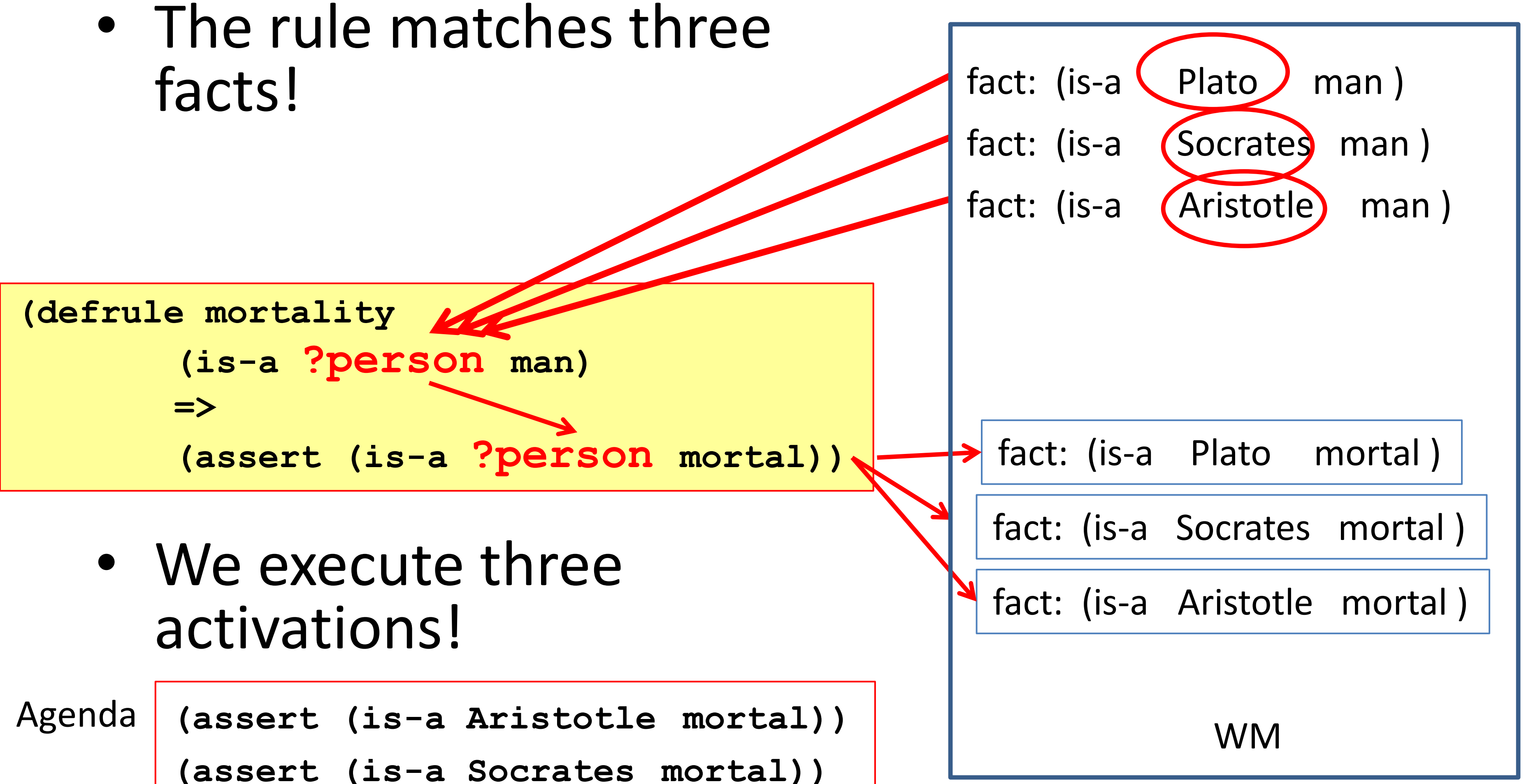
WM

- For each activation, we get **different bindings** of **?person**:
  - ?person = “Plato”
  - ?person = “Socrates”
  - ?person = “Aristotle”

Multiple Executions of a Rule

Match of Sets of Facts

- The rule matches three facts!



```
(defrule mortality
         (is-a ?person man)
        =>
         (assert (is-a ?person mortal))
```

fact: (is-a Plato man)

fact: (is-a Socrates man)

fact: (is-a Aristotle man)

fact: (is-a Plato mortal)

fact: (is-a Socrates mortal)

fact: (is-a Aristotle mortal)

WM

- We execute three activations!

Agenda

```
(assert (is-a Aristotle mortal))
(assert (is-a Socrates mortal))
(assert (is-a Plato mortal))
```

Pattern Matching

(3)Unordered/deftemplate Patterns – Slot-wise comparision

- Is there a match between the pattern and facts in WM?

```
(defrule married
    (person
                (name    ?name_1)
                (partner ?name_2))
    (person
                (name    ?name_2)
                (partner ?name_1))
   =>
    (assert (partners ?name_1 ?name_2))
)
```

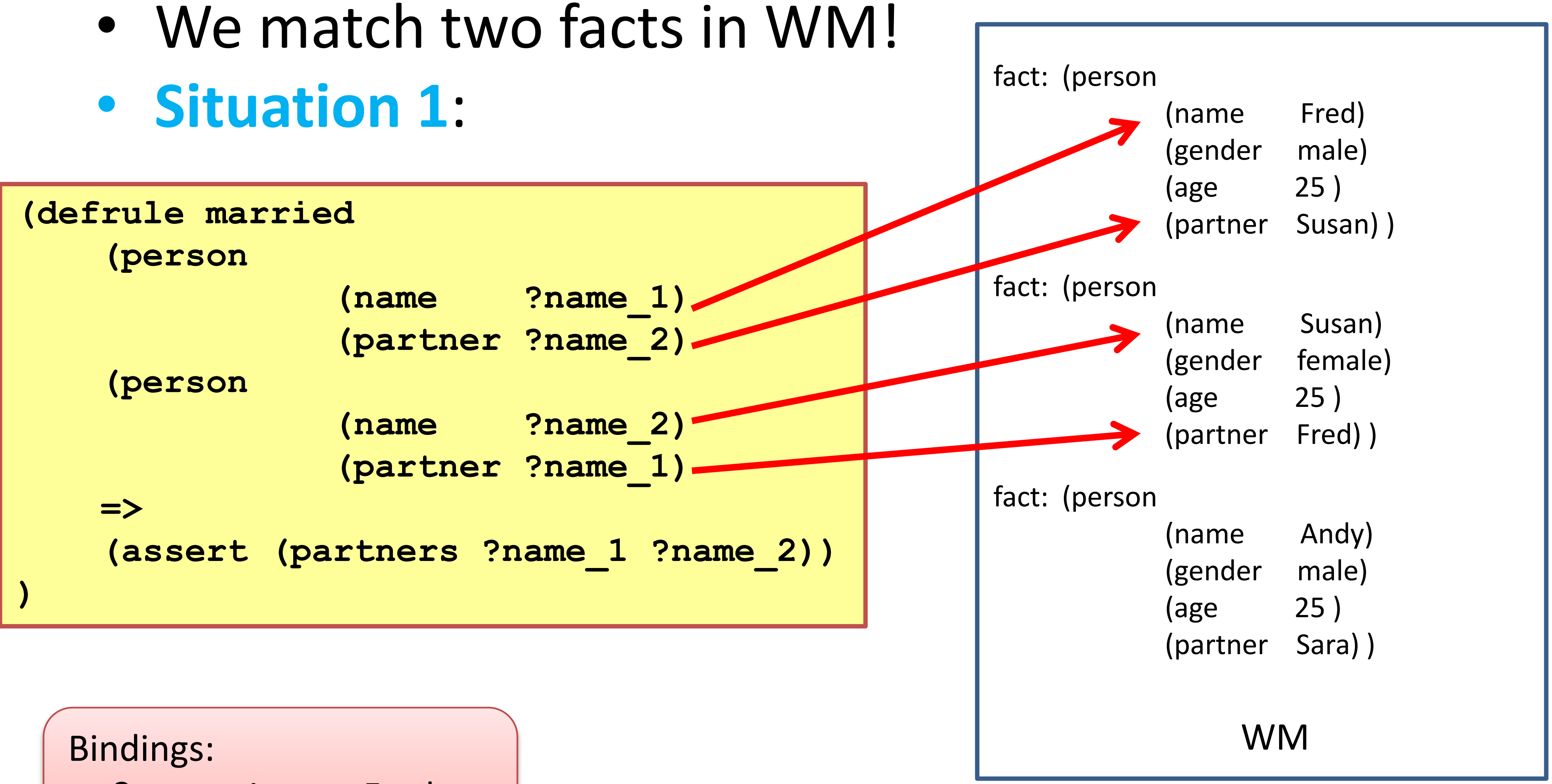
```
fact: (person
             (name      Fred)
             (gender    male)
             (age       25 )
             (partner   Susan) )

fact: (person
             (name      Susan)
             (gender    female)
             (age       25 )
             (partner   Fred) )

fact: (person
             (name      Andy)
             (gender    male)
             (age       25 )
             (partner   Sara) )

                 WM
```

Two conditions in the LHS

Pattern Matching

Unordered/deftemplate Patterns – Slot-wise comparision

- We match two facts in WM!
- **Situation 1**:



```
(defrule married
     (person
                  (name      ?name_1)
                  (partner ?name_2)
     (person
                  (name      ?name_2)
                  (partner ?name_1)
    =>
     (assert (partners ?name_1 ?name_2))
)
```

```
fact: (person
            (name       Fred)
            (gender     male)
            (age        25 )
            (partner    Susan) )
fact: (person
            (name       Susan)
            (gender     female)
            (age        25 )
            (partner    Fred) )
fact: (person
            (name       Andy)
            (gender     male)
            (age        25 )
            (partner    Sara) )
```

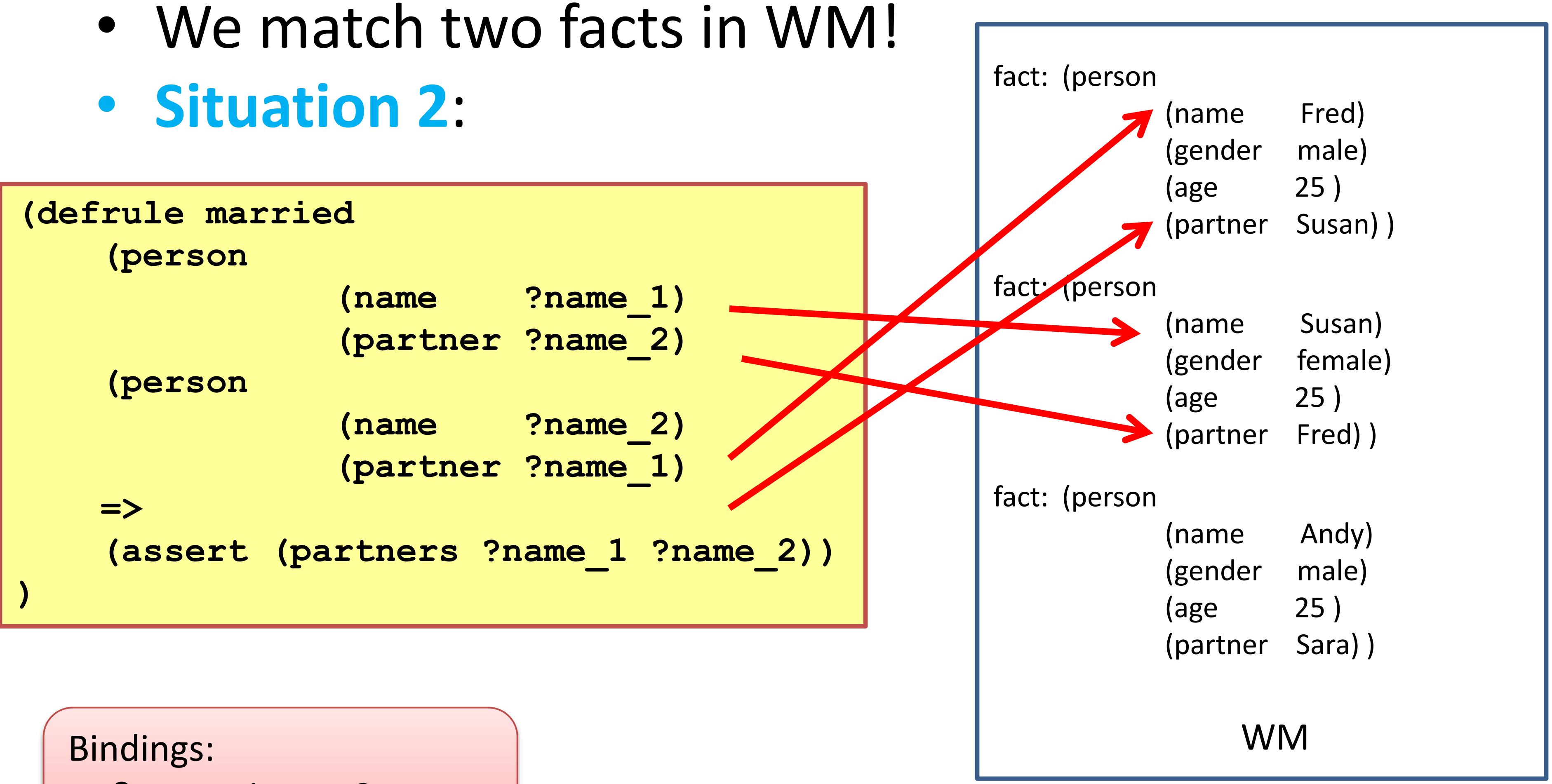
WM

Bindings:
?name_1 <--> Fred
?name_2 <--> Susan

Pattern Matching

Unordered/deftemplate Patterns – Slot-wise comparision

- We match two facts in WM!
- **Situation 2**:



```
(defrule married
    (person
                (name     ?name_1)
                (partner  ?name_2)
    (person
                (name     ?name_2)
                (partner  ?name_1)
    =>
    (assert (partners ?name_1 ?name_2))
)
```

```
fact:  (person
               (name       Fred)
               (gender     male)
               (age        25 )
               (partner    Susan) )
fact:  (person
               (name       Susan)
               (gender     female)
               (age        25 )
               (partner    Fred) )
fact:  (person
               (name       Andy)
               (gender     male)
               (age        25 )
               (partner    Sara) )
```

WM

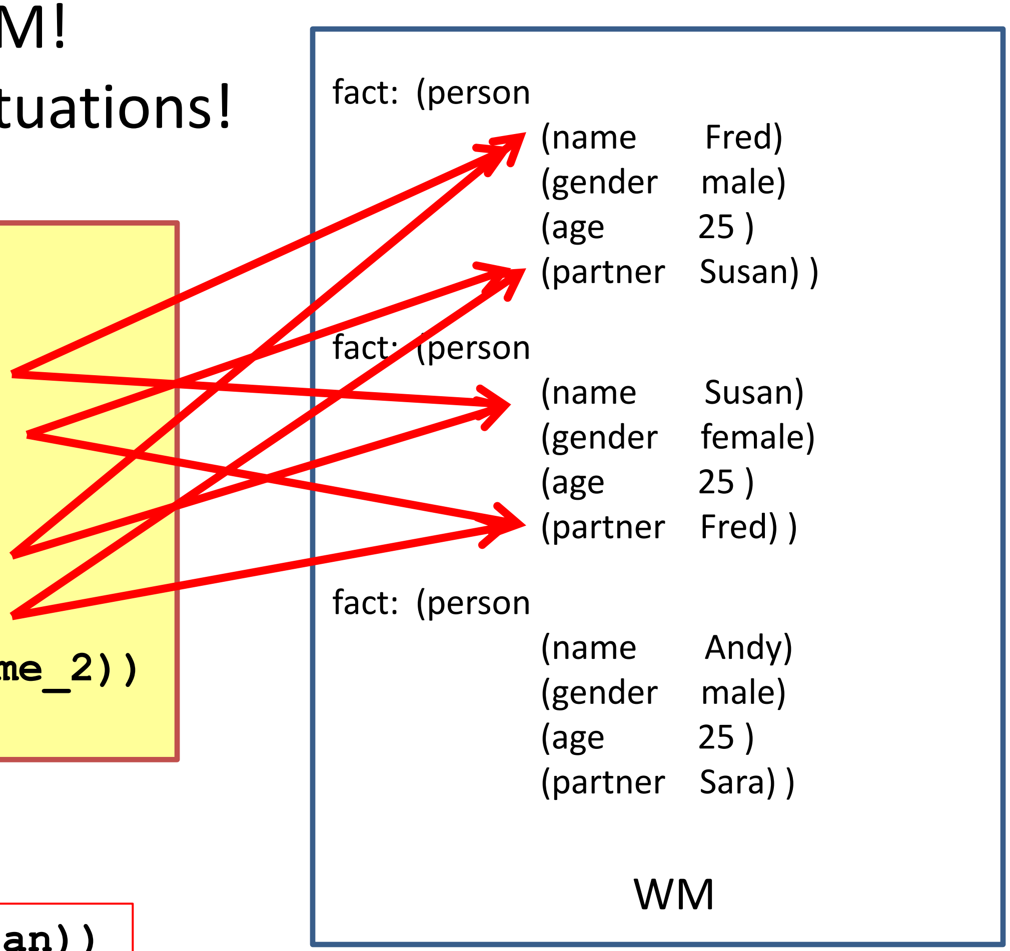
Bindings:
?name_1 <--> Susan
?name_2 <--> Fred

Rule Activation

Unordered/deftemplate Patterns – Slot-wise comparision

- We match two facts in WM!
- We have two matching situations!

```
(defrule married
     (person
                  (name     ?name_1)
                  (partner ?name_2)
     (person
                  (name     ?name_2)
                  (partner ?name_1)
    =>
     (assert (partners ?name_1 ?name_2))
)
```



```
fact: (person
          (name       Fred)
          (gender     male)
          (age        25 )
          (partner    Susan) )
fact: (person
          (name       Susan)
          (gender     female)
          (age        25 )
          (partner    Fred) )
fact: (person
          (name       Andy)
          (gender     male)
          (age        25 )
          (partner    Sara) )
```

- We get **two activations**!!

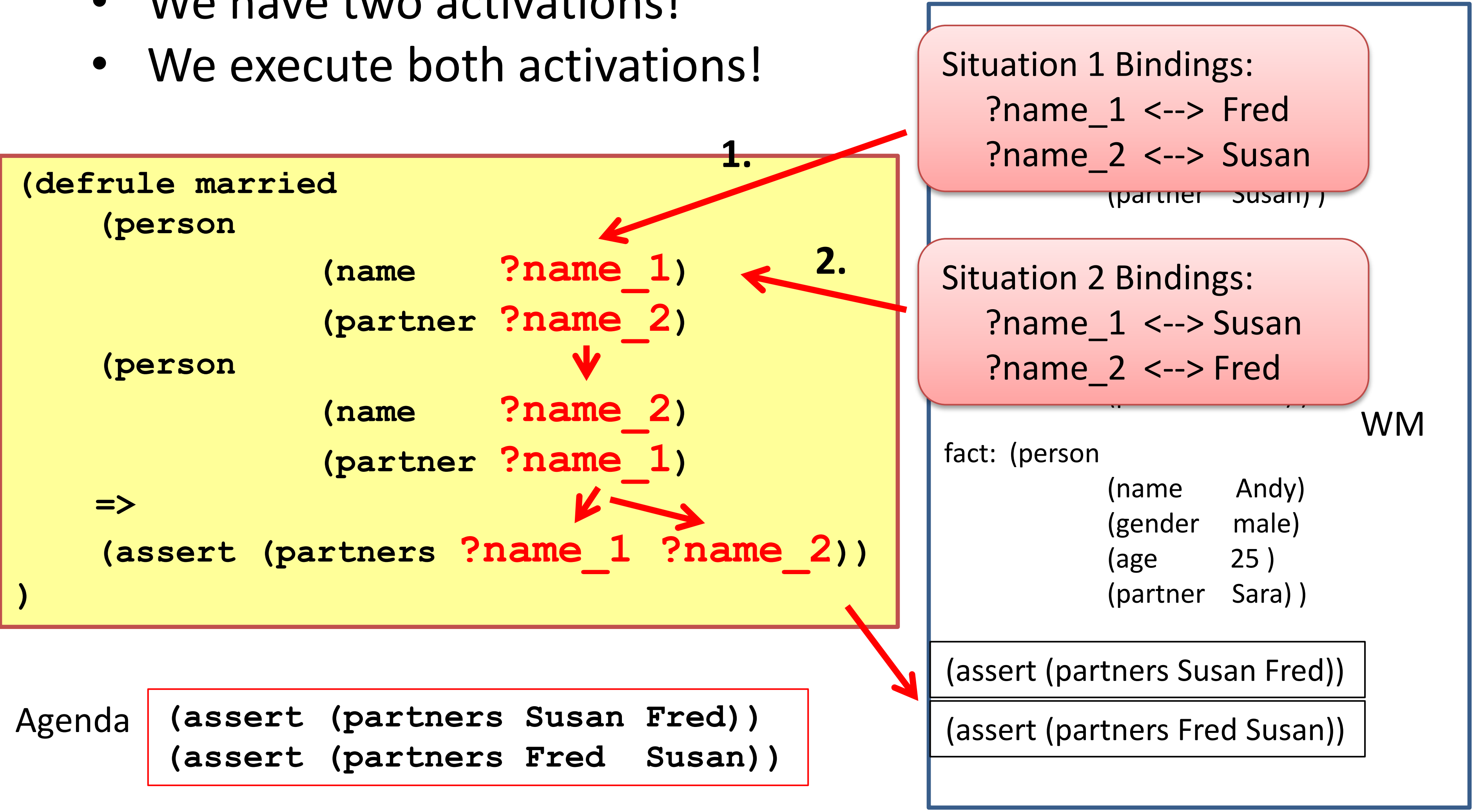
Agenda

```
(assert (partners Fred Susan))
(assert (partners Susan Fred))
```

Rule Execution

Unordered/deftemplate Patterns – Slot-wise comparision

- We have two activations!
- We execute both activations!



```
(defrule married
     (person
                 (name     ?name_1)
                 (partner  ?name_2)
     (person
                 (name     ?name_2)
                 (partner  ?name_1)
     =>
     (assert (partners ?name_1 ?name_2))
)
```

Situation 1 Bindings:
?name_1 <--> Fred
?name_2 <--> Susan

Situation 2 Bindings:
?name_1 <--> Susan
?name_2 <--> Fred

WM

(partner Susan))

fact: (person
(name Andy)
(gender male)
(age 25)
(partner Sara))

(assert (partners Susan Fred))

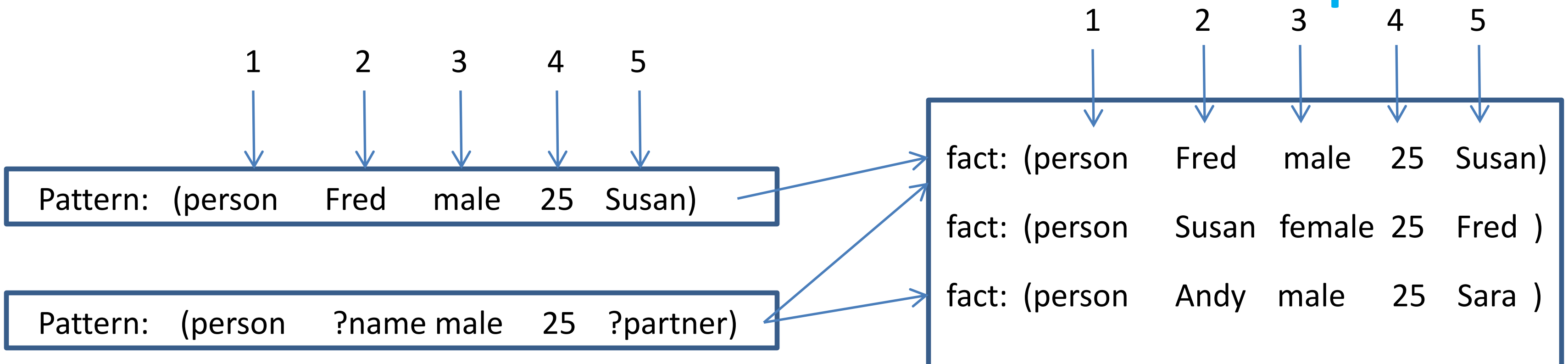
(assert (partners Fred Susan))

Agenda

```
(assert (partners Susan Fred))
(assert (partners Fred  Susan))
```

Pattern Matching Revisited

Ordered Patterns – Position-wise Comparision



1 2 3 4 5

Pattern: (person Fred male 25 Susan)

Pattern: (person ?name male 25 ?partner)

1 2 3 4 5

fact: (person Fred male 25 Susan)

fact: (person Susan female 25 Fred)

fact: (person Andy male 25 Sara)

What is Jess doing: Compares fields **position-wise**:

```
for each fact in WM:   (3)
   for each field in pattern:     (5)
      if field is a variable then
         assign value at same position in fact to variable
      if field is a constant then
         if field is not equal to value in fact then
            stop comparison (match failed, go to next fact)
   endfor
   if matching fact found, add rule to agenda
endfor
```

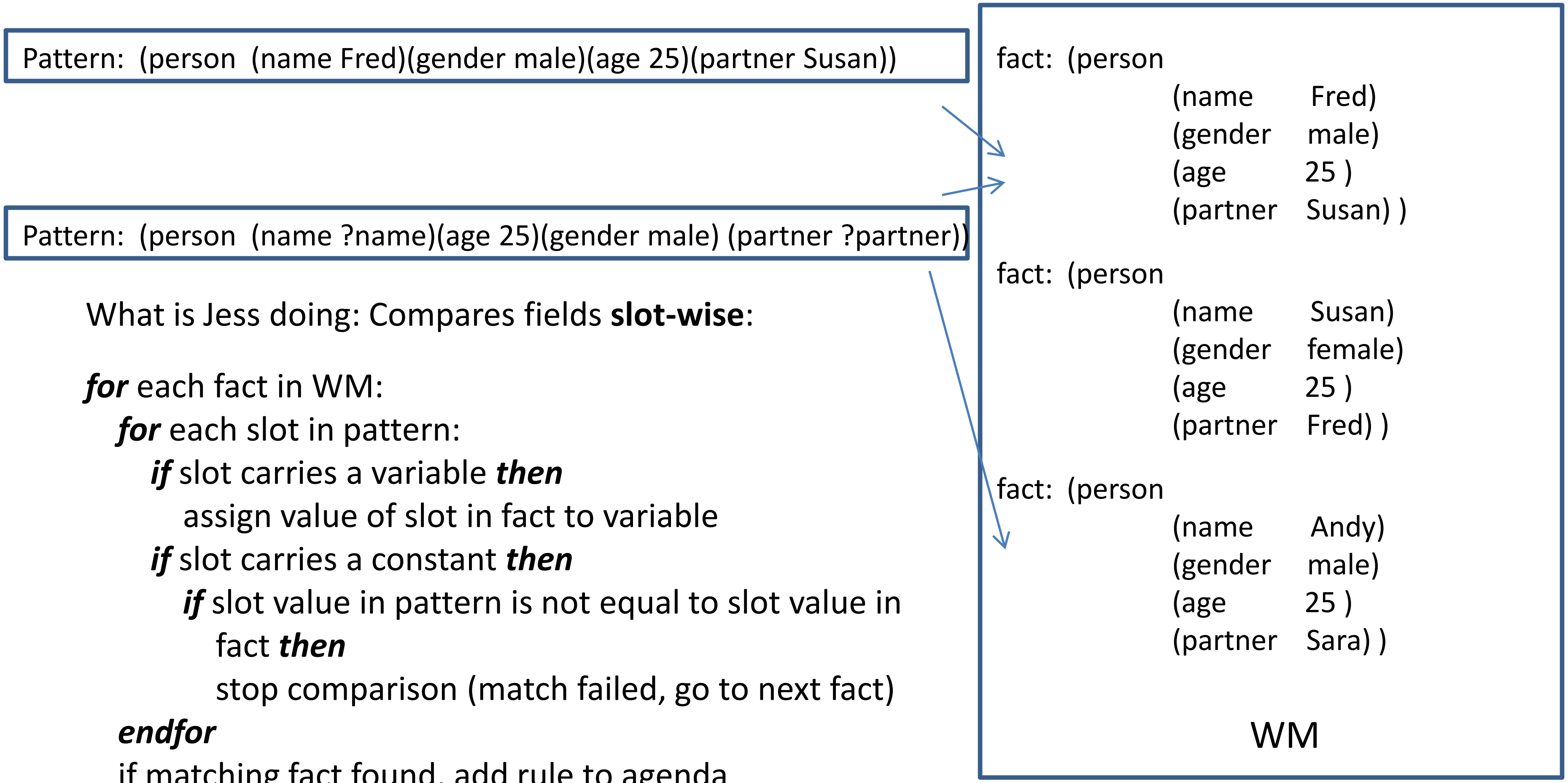
What is the difference?

How many time of comparisons does it take?

WM

Pattern Matching Revisited

Unordered/deftemplate Patterns – Slot-wise comparision



Pattern: (person (name Fred)(gender male)(age 25)(partner Susan))

Pattern: (person (name ?name)(age 25)(gender male) (partner ?partner))

```
fact: (person
              (name      Fred)
              (gender    male)
              (age       25 )
              (partner   Susan) )
fact: (person
              (name      Susan)
              (gender    female)
              (age       25 )
              (partner   Fred) )
fact: (person
              (name      Andy)
              (gender    male)
              (age       25 )
              (partner   Sara) )
```

WM

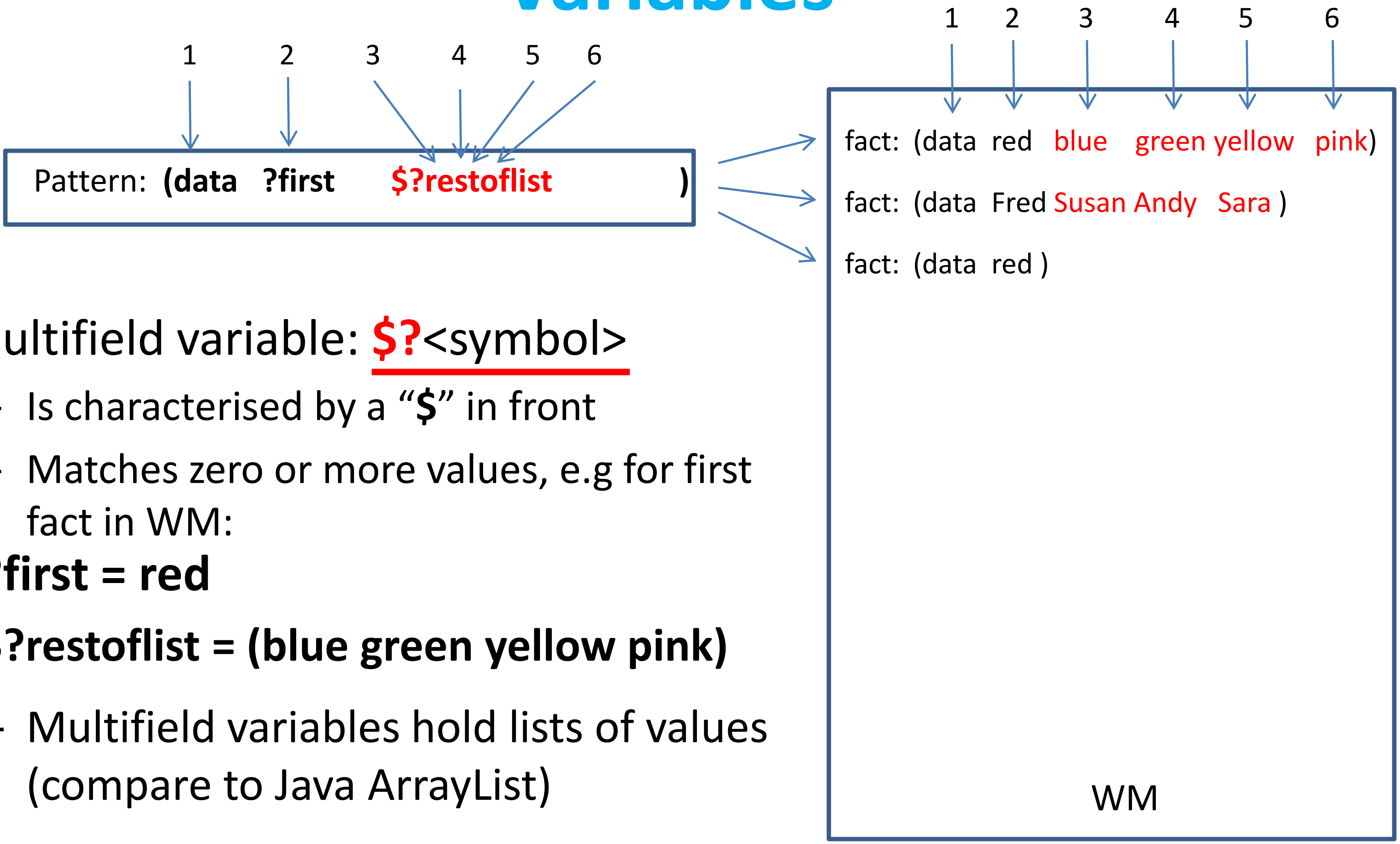
What is Jess doing: Compares fields **slot-wise**:

```
for each fact in WM:
   for each slot in pattern:
      if slot carries a variable then
         assign value of slot in fact to variable
      if slot carries a constant then
         if slot value in pattern is not equal to slot value in
            fact then
            stop comparison (match failed, go to next fact)
   endfor
   if matching fact found, add rule to agenda
endfor
```

Pattern matching: (4) Multifield Variables

- So far, we used simple variables like ?person that can be bound to single values
- Example
  - `(bind ?x 20)`
  - `(defrule mortality (is-a ?person man) => (assert (is-a ?person mortal))`
- Jess also has the concept of a multifield variable
  - Allows to manage lists of elements
  - Allows to match more than one element within ordered facts
  - Is comparable to a Vector / ArrayList in Java

Pattern Matching with **Multifield Variables**



Pattern: **(data ?first $?restoflist)**

fact: (data red blue green yellow pink)

fact: (data Fred Susan Andy Sara)

fact: (data red)

WM

- Multifield variable: **$?**<symbol>
  - Is characterised by a "**$**" in front
  - Matches zero or more values, e.g for first fact in WM:

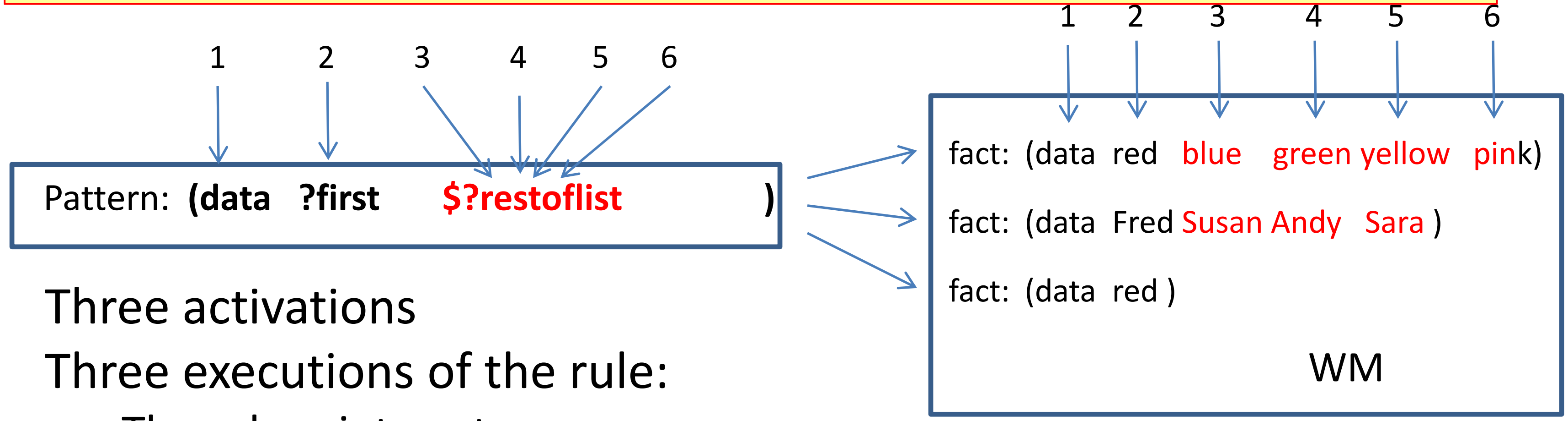
?first = red

$?restoflist = (blue green yellow pink)

  - Multifield variables hold lists of values (compare to Java ArrayList)

Pattern Matching with Multifield Variables

```
(defrule extract-list
         (data ?first $?restoflist )
         =>
         (printout t "the rest of the list: " ?restoflist crlf)
)
```



1 2 3 4 5 6

Pattern: **(data ?first $?restoflist)**

1 2 3 4 5 6

fact: (data red blue green yellow pink)

fact: (data Fred Susan Andy Sara)

fact: (data red)

WM

- Three activations
- Three executions of the rule:
  - The rule prints out:
    - 1.: **the rest of the list: blue green yellow pink**
    - 2.: the rest of the list: Susan Andy Sara
    - 3.: the rest of the list: □ !

NOTE: a multifield variable may also match "nothing", therefore may contain no binding!

Multifield Variables
LHS vs RHS – **Syntax** !

```
(defrule extract-list
        (data ?first $?restoflist )
        =>
        (printout t “the rest of the list: ” ?restoflist crlf)
)
```

Careful about Syntax !!

- LHS of a rule
  - the multifield variable is written with a “$” in front
- RHS of a rule (and any other part of a Jess program):
  - No distinction from **normal variables**

Pattern Matching: **(5) Wildcards**

- Wildcards are placeholders within patterns (like variables) that may match any element within a fact in WM
- Wildcards cannot bind values as they are not variables
- Two variants
  - Single-field wildcard
  - Multi-field wildcard

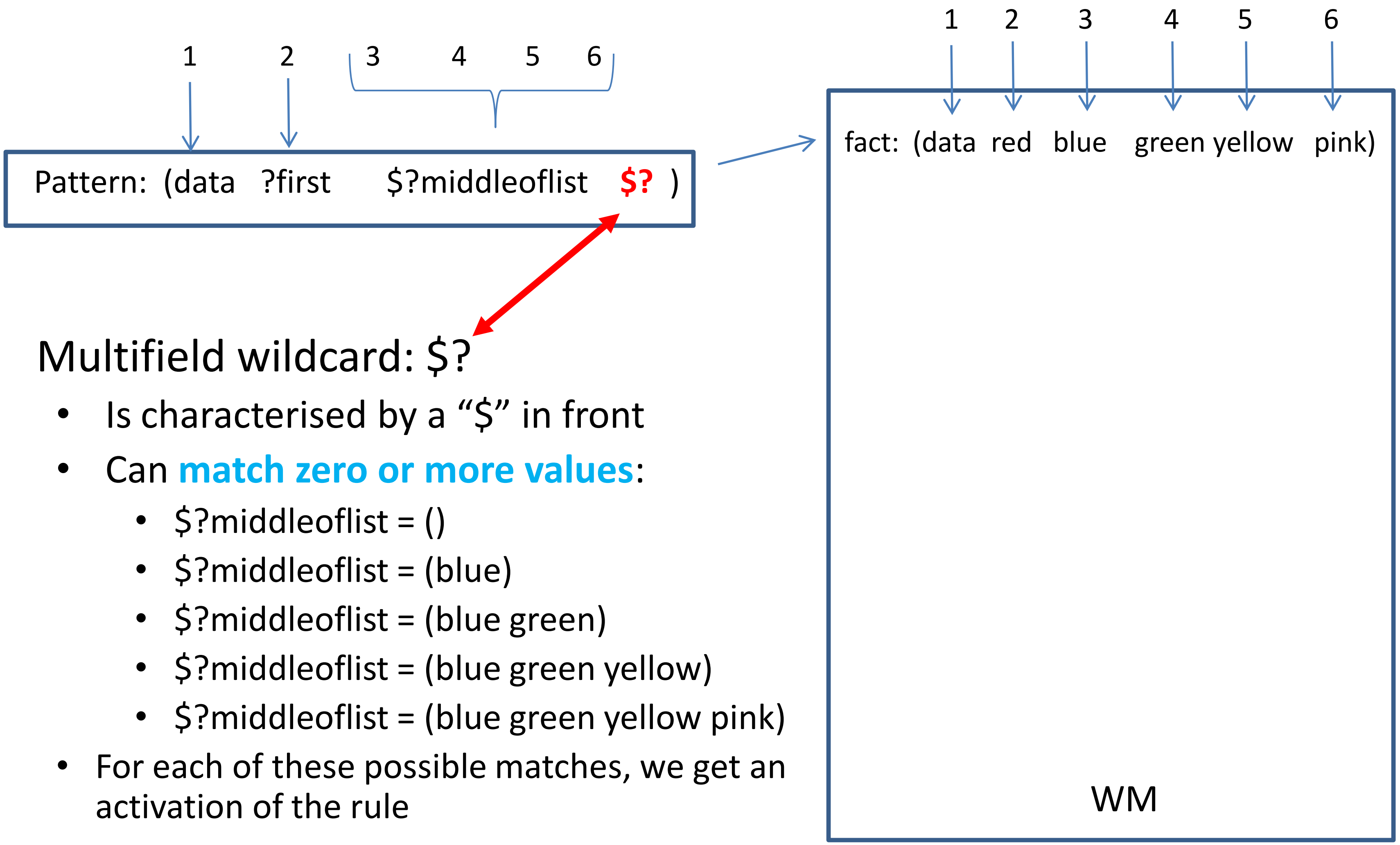
Pattern Matching with Wildcards

- **Single-field** wildcards in patterns:
  - written “**?**”
  - match any single-field value of a fact
  - Behaves similar to a variable in terms of matching (but without variable binding)
- **Multi-field** wildcards in patterns:
  - written “**$?**”
  - match zero or more fields in a fact.
  - Behaves similar to a multifield variable in terms of matching (but without variable binding)

- Example for using single-field wildcards:

```
(defrule exists-honest-person
       (found-honest ?)
       =>
       (printout t “Found at least one honest person!” crlf))
```

Pattern Matching with Wildcards



1 2 3 4 5 6

Pattern: (data ?first $?middleoflist $?)

1 2 3 4 5 6

fact: (data red blue green yellow pink)

WM

- Multifield wildcard: $?
  - Is characterised by a “$” in front
  - Can **match zero or more values**:
    - $?middleoflist = ()
    - $?middleoflist = (blue)
    - $?middleoflist = (blue green)
    - $?middleoflist = (blue green yellow)
    - $?middleoflist = (blue green yellow pink)
  - For each of these possible matches, we get an activation of the rule

Example Multifield Wildcards

```
(deffacts colors
        (data red blue green yellow pink))
(defrule r1
        (data ?first $?middleoflist $? )
        =>
        (printout t "first = " ?first ", middleoflist = " ?middleoflist crlf))
```

Careful about Syntax !!

LHS

RHS

```
Jess> (facts)
f-0   (MAIN::initial-fact)
f-1   (MAIN::data red blue green yellow pink)
For a total of 2 facts in module MAIN.
Jess> (agenda)
[Activation: MAIN::r1  f-1 ; time=2 ; totalTime=2 ; salience=0]
[Activation: MAIN::r1  f-1 ; time=2 ; totalTime=2 ; salience=0]
[Activation: MAIN::r1  f-1 ; time=2 ; totalTime=2 ; salience=0]
[Activation: MAIN::r1  f-1 ; time=2 ; totalTime=2 ; salience=0]
[Activation: MAIN::r1  f-1 ; time=2 ; totalTime=2 ; salience=0]
For a total of 5 activations in module MAIN.
Jess> (run)
first = red, middleoflist = ()
first = red, middleoflist = (blue green yellow pink)
first = red, middleoflist = (blue green yellow)
first = red, middleoflist = (blue green)
first = red, middleoflist = (blue)
5
Jess>
```

Example Matching with Wildcards

- A single-field wildcard “?” matches any single field:

Pattern: `(is-a toy ? yellow )`

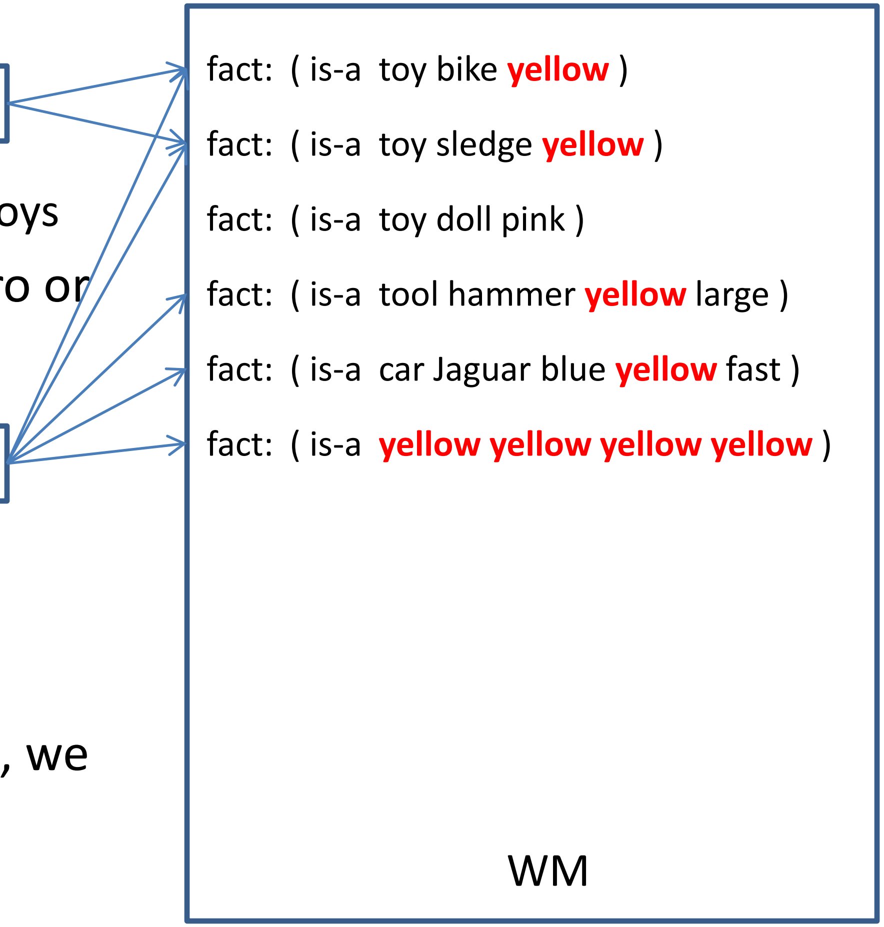
  - (is-a toy ? yellow) matches all yellow toys

- A multifield wildcard “$?” matches zero or more fields:

Pattern: `(is-a $? yellow $? )`

  - (is-a $? yellow $?) matches any “is-a” fact containing “yellow”
  - How many rule activations do we get?

- Observation: with multifield wildcards, we can “**traverse**” facts and generate rule activations for each match



fact: (is-a toy bike **yellow**)

fact: (is-a toy sledge **yellow**)

fact: (is-a toy doll pink)

fact: (is-a tool hammer **yellow** large)

fact: (is-a car Jaguar blue **yellow** fast)

fact: (is-a **yellow yellow yellow yellow**)

WM

Example Matching with Wildcards

```
(defrule how-many-yellows
         (is-a $? yellow $?)
         =>
         (printout t "Found yellow" crlf)
)
(assert (is-a yellow yellow yellow yellow))
TRUE
Jess> TRUE
Jess> <Fact-0>
Jess> (facts)
f-0   (MAIN::is-a yellow yellow yellow yellow)
For a total of 1 facts in module MAIN.
Jess> (agenda)
[Activation: MAIN::how-many-yellows  f-0 ; time=1 ; totalTime=1 ; salience=0]
[Activation: MAIN::how-many-yellows  f-0 ; time=1 ; totalTime=1 ; salience=0]
[Activation: MAIN::how-many-yellows  f-0 ; time=1 ; totalTime=1 ; salience=0]
[Activation: MAIN::how-many-yellows  f-0 ; time=1 ; totalTime=1 ; salience=0]
For a total of 4 activations in module MAIN.
Jess>
```

Conditional Elements

Rule Condition (LHS)

Rule:
LHS
ANTECENT
=>
RHS
CONSEQUENT

- Remember:
  - The LHS contains patterns that decide whether a rule is activated
  - The LHS as well as parts of it are regarded as “**condition elements**” (elements of the condition of the rule)
- So far:
  - We have constructed rules, where the LHS is comprised of a list of patterns:
    - **This is a conjunction of patterns**

```
(defrule creature-species-1
        (order-of carnivore)
        (colour-of tawny)
        (marking-of black-stripes)
  =>
   (printout t “Observed species is a Tiger!”
crlf)
)
```

IF the first pattern **AND**
the second pattern **AND**
the third pattern
matches facts in WM
THEN activate the rule

Rule Condition – Disjunction

Conditional Element "or"

```
( or <condition-element-1> <condition-element-2> ... )
```

- More complex rule conditions – **Disjunction** over condition elements of the LHS, using the operator "**or**"
- Meaning:
  - If any one of the condition elements matches facts in WM, then the rule becomes activated
- Example:

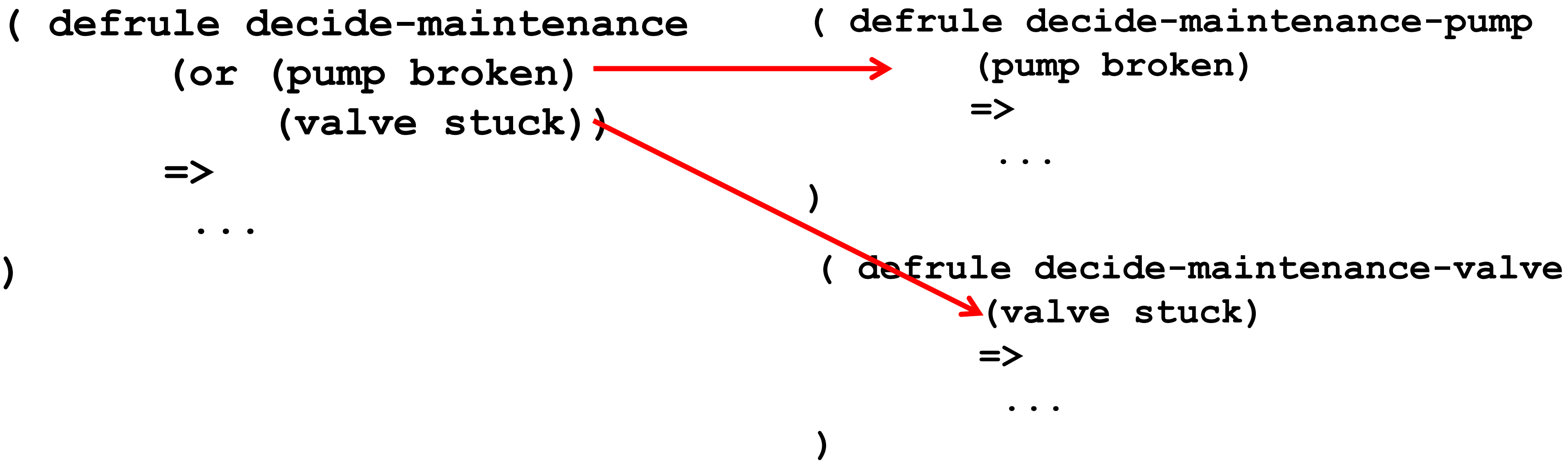
```
( defrule decide-maintenance
        (or  (pump broken)
             (valve stuck))
        =>
         ...
)
```

IF the first pattern **OR**
the second pattern
matches facts in WM
THEN activate the rule

Rule Conditions – Disjunction

Conditional Element “or”

- We could also write a **separate rule** for each or-condition element



```
( defrule decide-maintenance
        (or (pump broken)
            (valve stuck))
      =>
        ...
)
```

```
( defrule decide-maintenance-pump
        (pump broken)
        =>
         ...
)
```

```
( defrule decide-maintenance-valve
        (valve stuck)
        =>
         ...
)
```

Rule Condition – Conjunction

Conditional Element “and”

```
( and <condition-element-1> <condition-element-2> ... )
```

- Conjunction over patterns / condition elements, using the logical operator “and”
- Remark:
  - A LHS of a rule comprised of a list of condition elements is **implicitly** regarded as a conjunction – no need to write an explicit “and”
- This operator is needed for more complex expressions over condition elements of the LHS:

```
( defrule is-it-working
      (or (fuse checked)
          (and (switched on)(plugged in))
      =>
       ...
)
```

Complex conditions, how to make it more efficient?

Rule Condition - Negation

```
( not (<condition-element>) )
```

- A rule fires if something “is not the case”
  - More precise: the rule fires because no matching fact is found in WM
- We check whether something is **not** in WM
  - `(not (person (gender female) (age over60) ) )`
  - There is no female over 60

- Note: Closed-World Assumption:
  - If something is not known to be true (not in WM) it is assumed not to be true
  - Not necessarily the case – it is not recorded as a fact inside our software system, but it could still be true (there are many females over 60 in this world!)

Conditional Element - **test**

```
( test (<function> <parameter-1> <parameter-2> ... ))
```

- Jess provides the construct “test” to apply **built-in functions**
- Most important functions:
  - For numbers: = <> > < >= <=
  - For strings: eq neq gt lt ge le

```
(defrule age-test
    (person (name ?person1) (age ?age1))
    (person (name ?person2) (age ?age2))
    (test (neq ?person1 ? person2))
    (test (> ?age1 ?age2))
   =>
    (printout t ?person1 “is older than” ?person2 crlf)
)
```

- “test” cannot be the first Condition element of a LHS, tests have to be added to the very end of the LHS !

Conditional Element - **exists**

```
( exists (<condition-element>) )
```

- The condition element “exists” is true if there exist any facts that match the enclosed pattern:

```
(defrule exists-an-honest-man
    (exists (honest ?))
    =>
    (printout t "There is at least one honest man!" crlf)
)
```

- This is equivalent to the following rule – “exists” is a shortcut for double negation:

```
(defrule exists-an-honest-man
    (not (not (honest ?)))
    =>
    (printout t "There is at least one honest man!" crlf)
)
```

Constraints

- Constraints can be defined over variables
  - Define what values a variable is allowed to hold
- Constraints influence the pattern matching
  - Constraints over variables within a pattern determine which facts are matched by this pattern

Connective Constraints

constraint on variable ?y

“and” “not”

```
(defrule isnot-clear
     (goal clear ?x)
     (block (label ?x)(has-on-top ?y & ~nothing))
    =>
     (printout t "new sub-goal: clear " ?y crlf)
     (assert (goal clear ?y)))
```

- Variable **?y** is constrained:
  - It will match anything except the String “**nothing**”
  - Symbol “**&**” can be read as a logical “and” between the value held by **?y** and the connected constraint

Connective Constraints

- We can connect constraints to variables or to each other:
  - **&** : represents “and”, satisfied if both adjoining constraints are satisfied
  - **|** : represents “or”: satisfied if one of the adjoining constraints is satisfied
  - **~** : represents “not”: satisfied if the following constraint is not satisfied
  - Precedence: **~ & |**
- Example patterns:

```
(is-a toy ?name ?color & ~blue &~ red)

(person (name ?name
          (occupation ?myjob & academic | accountant )
```

?myjob can have two possible bindings:
- either “academic” or “accountant”

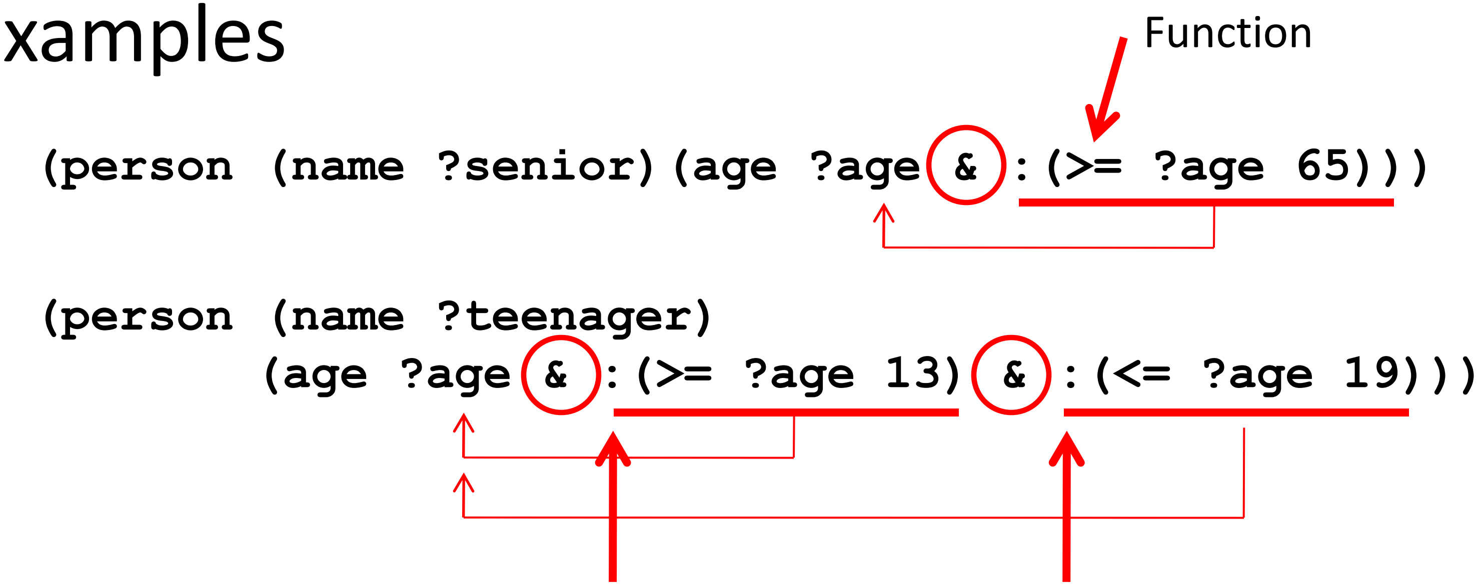
Predicate Field Constraints

Using **Functions** within Constraints

- Satisfied if the value returned by a function is not False

```
:( <function> <parameter-1> … )
```

- Examples



Function

```
(person (name ?senior)(age ?age & :(>= ?age 65)))

(person (name ?teenager)
        (age ?age & :(>= ?age 13) & :(<= ?age 19)))
```

returns TRUE or FALSE, checks a property of the constrained variable

Manipulating Lists

Manipulating Lists

- Jess allows us to explicitly create new lists (and, e.g. assert them as new facts) or manipulate these lists
- The following functions are available:
  - "**length$**": returns the **number** of items in a list
  - "**member$**": checks whether an item is part of a list
  - "**create$**": used to create a list of items
  - "**nth$**": retrieve the nth element of list
  - "**first$**": returns a new list containing only the first element of the list (creates a new list)
  - "**rest$**": returns a new list without the first element of the list (creates a new list)
  - "**insert$**": returns a new list with a new element inserted at a given index (creates a new list)
- Look up the Jess manual: http://www.jessrules.com/jess/docs/71/

Jess deals with ***list*** instead of ***string***, why?

Manipulating Lists

- Create a list and bind it to a variable:

```
Jess> (bind ?grocery-list (create$ eggs bread milk))
(eggs bread milk)
```

- Access the n-th element in the list:

```
Jess> (printout t (nth$ 2 ?grocery-list) crlf)
bread
```

- Get first and rest of list:

```
Jess> (first$ ?grocery-list)
(eggs)
```

```
Jess> (rest$ ?grocery-list)
(bread milk)
```

Manipulating Lists

- Extend a list, add items

```
Jess> (bind ?grocery-list (create$ eggs bread milk))
(eggs bread milk)

Jess> (bind ?grocery-list (create$ ?grocery-list salt soap))
(eggs bread milk salt soap)
Jess>
```

- Insert an item into a list

```
Jess> (bind ?grocery-list (insert$ ?grocery-list 3 beans))
(eggs bread beans milk salt soap)
Jess>
```

Manipulating Lists

- Use list manipulation in rules:

```
(defrule manipulate-list
    (grocery-list $?grocery-list)
    =>
    (printout t (nth$ 2 ?grocery-list) crlf)
    (printout t (first$ ?grocery-list) crlf)
    (printout t (rest$ ?grocery-list)  crlf)
    (bind ?grocery-list (create$ ?grocery-list salt soap))
    (printout t ?grocery-list crlf)
)
```

Careful about Syntax !!

- Note the syntax of multifield variables:
    - LHS: “**$?**” before name
    - RHS and inside constraints, functions etc.: only “**?**” !!

Constraints and and Lists

- “**length$**”: returns the length of the item list held by a multifield variable
- “**member$**”: used to check whether an item is part of a list held by a multifield variable

Careful about Syntax !!

```
(defrule large-order-and-no-milk
   (shopping-cart
        (customer-id   ?id)
        (contents      $?cart & :(and (> (length$ ?cart) 50)
                                      (not (member$ milk ?cart)))
   =>
   (printout t “Do you need milk?” crlf))
```

- Example: automated shopping advice: “if the customer shops more than 50 items and there is no milk in the cart, ask the customer”
- Look up the Jess manual: http://www.jessrules.com/jess/docs/71/

Writing LHS with constraints

```
(defrule test-age-1
   (person (name ?name1) (age ?age1))
   (person (name ?name2) (age ?age2))
   (test (neq ?person1 ?person2))
   (test (> ?age1 ?age2))
   =>
   (printout t ?name1 "is older than " ?name2 crlf))
```

- This is the same:

```
(defrule test-age-2
   (person (name ?person1) (age ?age1))
   (person (name ?person2 &:(neq ?person1 ?person2)
           (age  ?age2 &:(> ?age1 ?age2)))
 =>
   (printout t ?name1 "is older than " ?name2 crlf))
```

Summary

- When and how to activate a rule?
  - Pattern matching
  - Conditions
  - List (not string)

- Question?