Jess Efficiency

The Jess Language Part 4

CS3025, Knowledge-Based Systems

Lecture 13

Yuting Zhao

Yuting.zhao@gmail.com

2017-10-31

Questions from past lectures

- Jess university licence:
- https://drive.google.com/file/d/0Bz3MNdYF1WaOYmVUY0d1SF9Gd2M/view?usp=sharing

Jess Efficiency

- Revisit Pattern Matching
- Relationships between Patterns
- Rete networks
- Optimization
- Handling “or”, negation

Revisit Pattern Matching

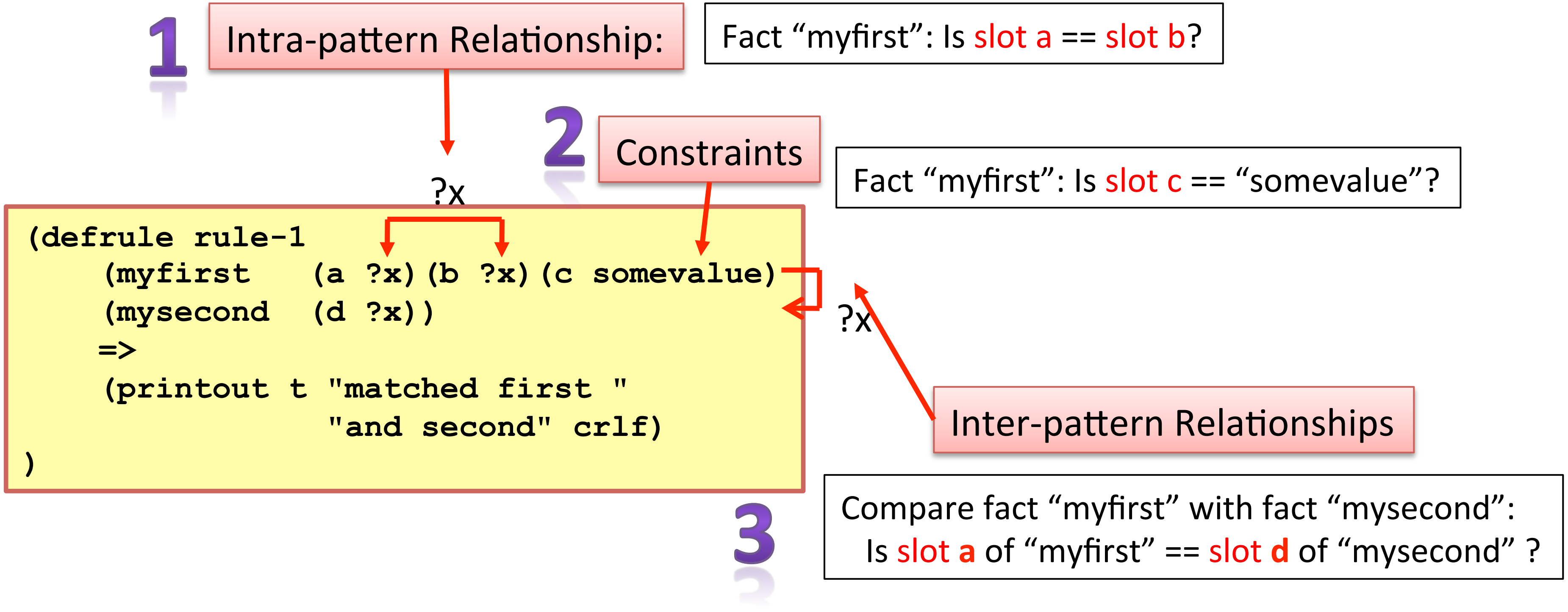
- What we know so far
  - An expert system consists of **rules** and a set of **facts** that are manipulated by these rules
  - If a new fact is inserted, the LHS of rules possibly match, which would lead to an **activation** of these rules
  - Therefore: All rules have to be matched against all facts each time a new fact is inserted (**check all rules**)

Revisit Pattern Matching

- **Naïve approach**
  - Match each pattern of each rule against each fact in WM at each execution cycle
  - This is very costly
- We need a more efficient approach, where redundant repeated matches are avoided
- **RETE**
  - Is an algorithm that compiles the LHS of rules into a network of nodes that represent matching tests for facts
  - When facts are asserted into WM, they are filtered through this network and at each node, these tests are applied to them
  - Rete is efficient because it stores partial matching results to avoid repetitions of the same matches

Relationships between Patterns

```
(deftemplate myfirst
              (slot a)(slot b)(slot c))
(deftemplate mysecond
              (slot d)(slot e))
(deftemplate mythird
              (slot f))
```



1 Intra-pattern Relationship:

Fact “myfirst”: Is slot a == slot b?

2 Constraints

Fact “myfirst”: Is slot c == “somevalue”?

```
(defrule rule-1
    (myfirst   (a ?x)(b ?x)(c somevalue))
    (mysecond  (d ?x))
    =>
    (printout t "matched first "
                "and second" crlf)
)
```

Inter-pattern Relationships

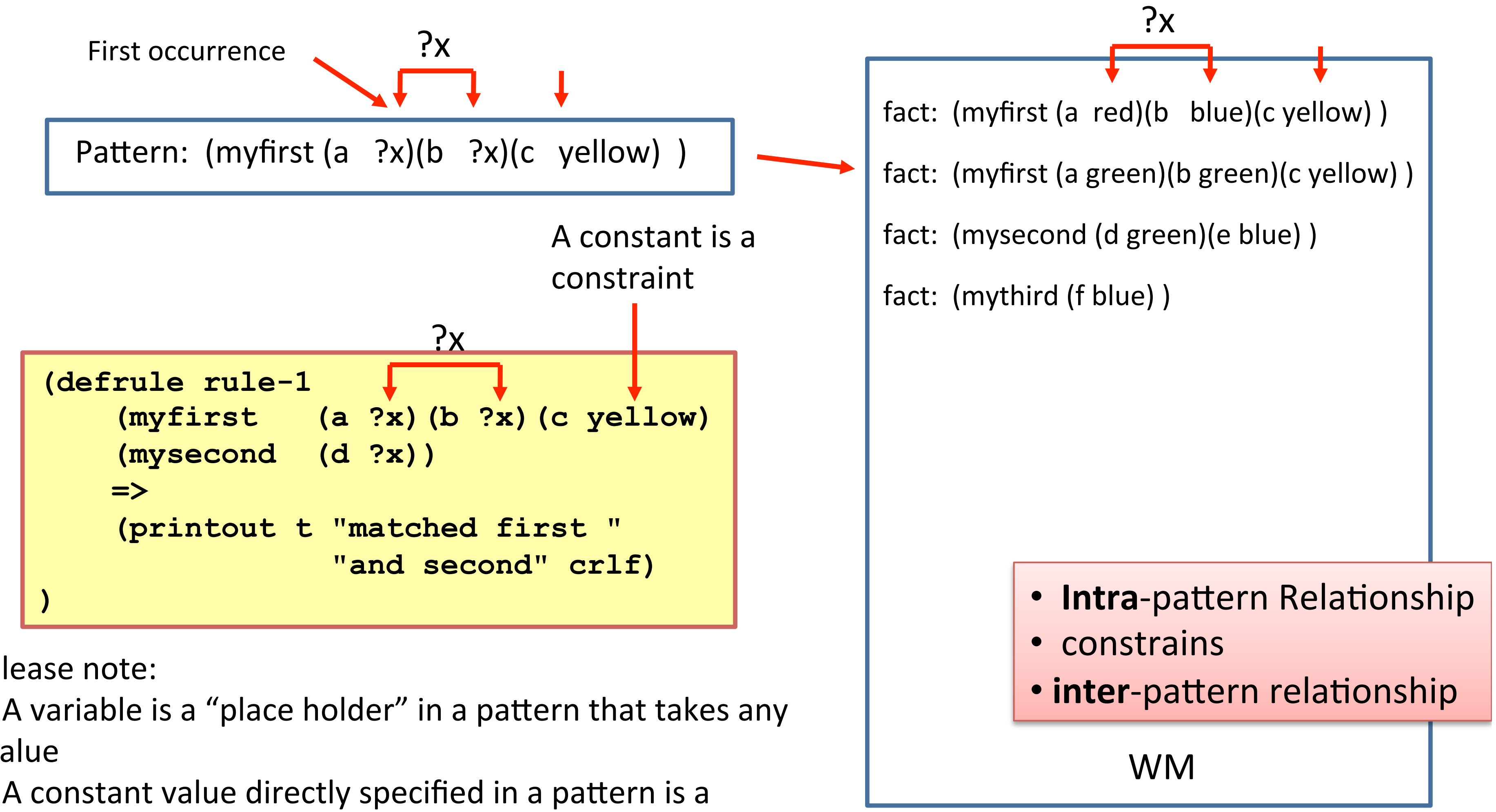
3

Compare fact “myfirst” with fact “mysecond”:
Is slot **a** of “myfirst” == slot **d** of “mysecond” ?

Relationships between Patterns

- We have to pay particular attention to variables
  - A **variable** can occur multiple times within a pattern – intra-pattern relationships:
    - It will receive a **binding** at its first occurrence (most left occurrence in pattern)
    - This binding will determine throughout the rest of the pattern what facts this pattern will match
  - A **variable** can occur in more than one pattern of a LHS of a rule – inter-pattern relationships:
    - It will receive a binding at its first occurrence (most left occurrence in first pattern)
    - This binding will determine throughout the complete LHS of a rule, what facts are matched by the patterns

Example: Intra-Pattern and Constants/ Constraints



Pattern: (myfirst (a ?x)(b ?x)(c yellow))

```
(defrule rule-1
    (myfirst   (a ?x)(b ?x)(c yellow)
    (mysecond  (d ?x))
    =>
    (printout t "matched first "
                "and second" crlf)
)
```

fact: (myfirst (a red)(b blue)(c yellow))

fact: (myfirst (a green)(b green)(c yellow))

fact: (mysecond (d green)(e blue))

fact: (mythird (f blue))

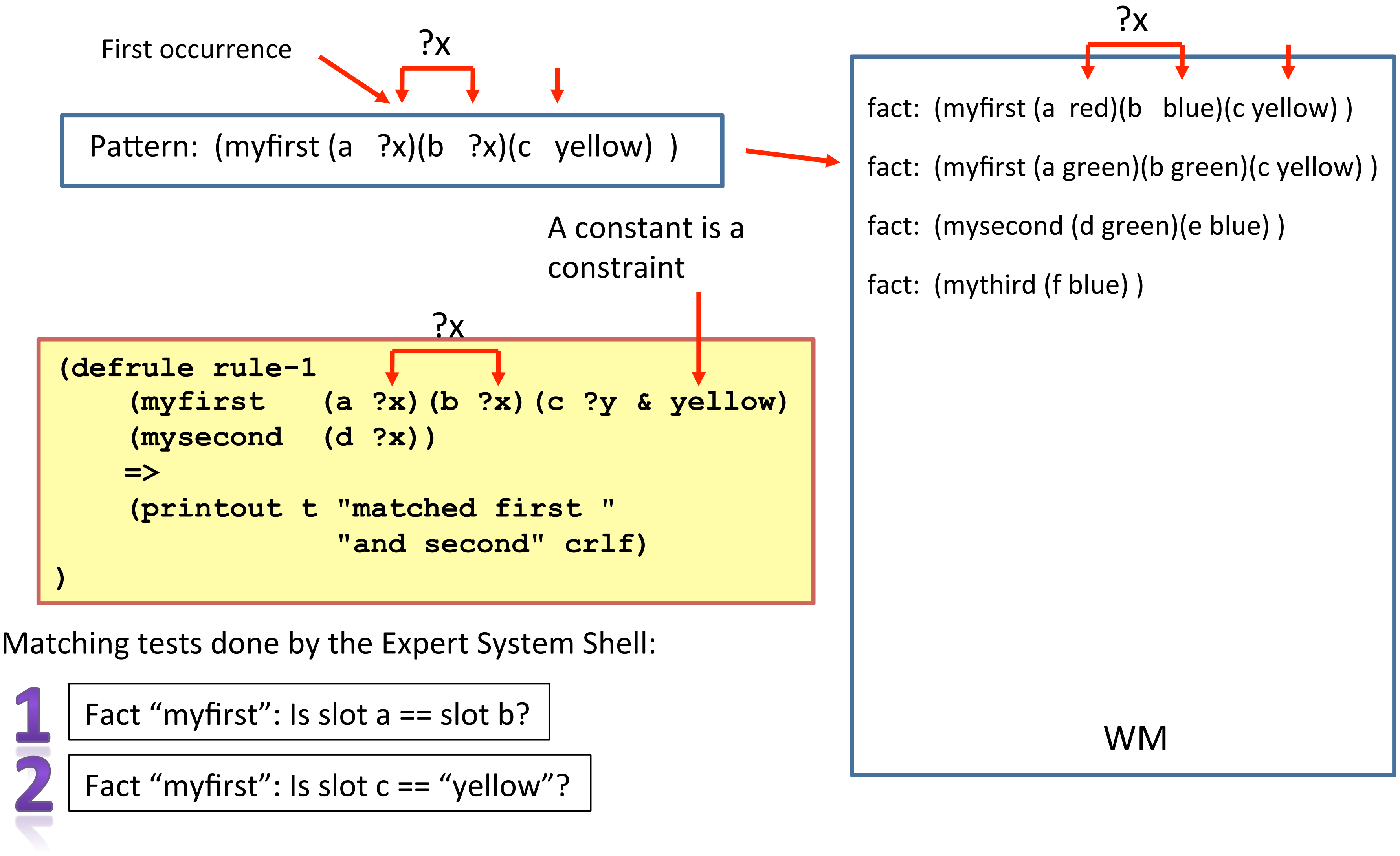
- **Intra**-pattern Relationship
- constrains
- **inter**-pattern relationship

Please note:

- A variable is a “place holder” in a pattern that takes any value

- A constant value directly specified in a pattern is a “constraint” – it constrains what can be matched by this pattern

Example: Intra-Pattern and Constants/ Constraints



First occurrence

?x

Pattern: (myfirst (a ?x)(b ?x)(c yellow))

A constant is a constraint

?x

```
(defrule rule-1
    (myfirst   (a ?x)(b ?x)(c ?y & yellow)
    (mysecond  (d ?x))
    =>
    (printout t "matched first "
                "and second" crlf)
)
```

?x

fact: (myfirst (a red)(b blue)(c yellow))

fact: (myfirst (a green)(b green)(c yellow))

fact: (mysecond (d green)(e blue))

fact: (mythird (f blue))

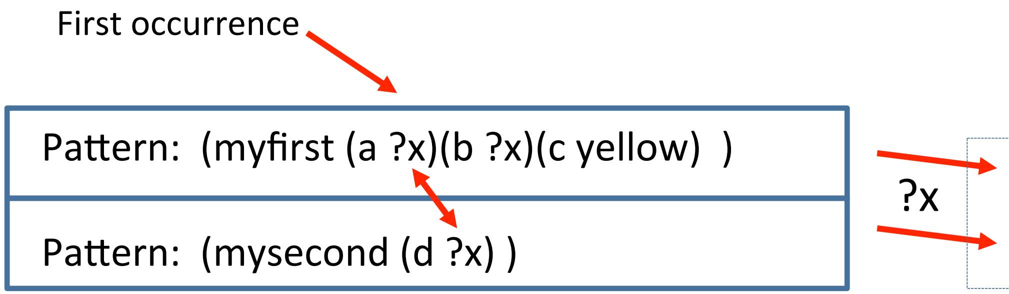
WM

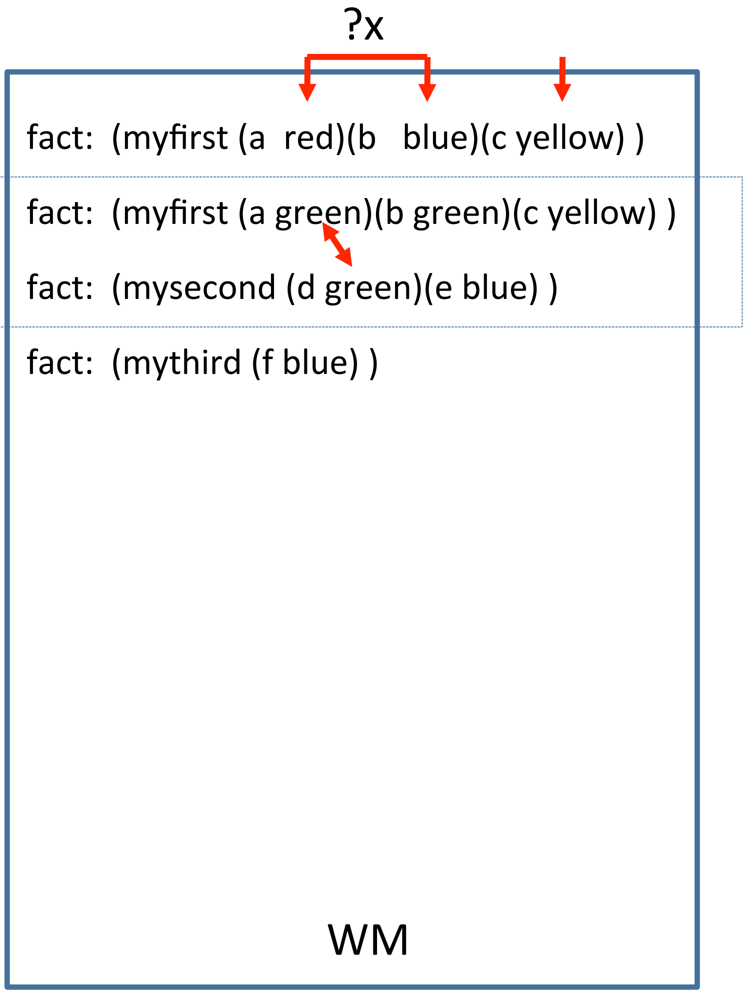
Matching tests done by the Expert System Shell:

1 Fact “myfirst”: Is slot a == slot b?

2 Fact “myfirst”: Is slot c == “yellow”?

Example: Inter-Pattern


First occurrence
Pattern: (myfirst (a ?x)(b ?x)(c yellow))
Pattern: (mysecond (d ?x))
?x


?x
fact: (myfirst (a red)(b blue)(c yellow))
fact: (myfirst (a green)(b green)(c yellow))
fact: (mysecond (d green)(e blue))
fact: (mythird (f blue))
WM

```
(defrule rule-1
    (myfirst   (a ?x)(b ?x)(c yellow)
    (mysecond  (d ?x))
    =>
    (printout t "matched first "
                "and second" crlf)
)
```

?x

Matching tests done by the Expert System Shell:

3 Compare a fact “myfirst” with a fact “mysecond”:
Is slot **a** of “myfirst” == slot **d** of “mysecond” ?

Rete Networks

- Jess uses a **Rete Network** for efficient pattern matching
- RETE
  - Is an algorithm that compiles the LHS of rules into a network of nodes that represent matching tests for facts
  - When facts are asserted into WM, they are filtered through this network and at each node, these tests are applied to them

Compile Rule into a Rete Network

```
(deftemplate myfirst
             (slot a)(slot b)(slot c))
(deftemplate mysecond
             (slot d)(slot e))
(deftemplate mythird
             (slot f))
```

Intra-pattern Relationship

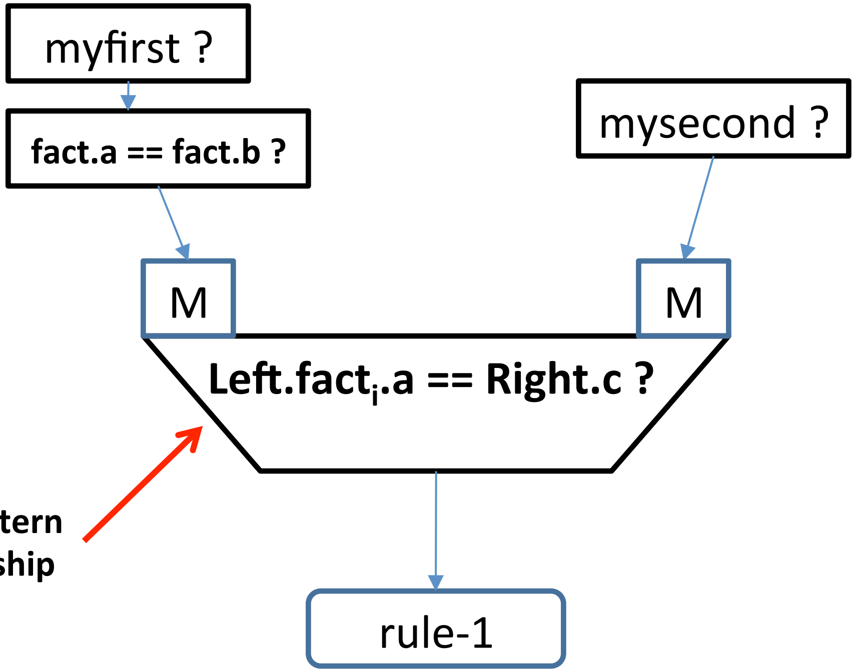
?x

```
(defrule rule-1
    (myfirst   (a ?x)(b ?x)
    (mysecond  (c ?x))
    =>
    (printout t "matched first "
                "and second" crlf)
)
```

?x

Inter-pattern Relationship

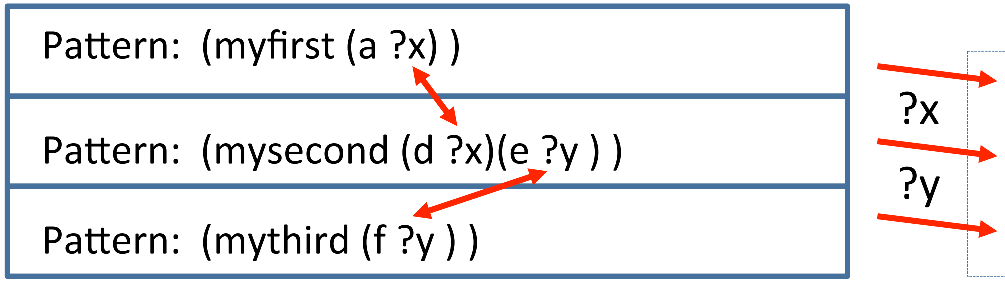
Rule rule-1

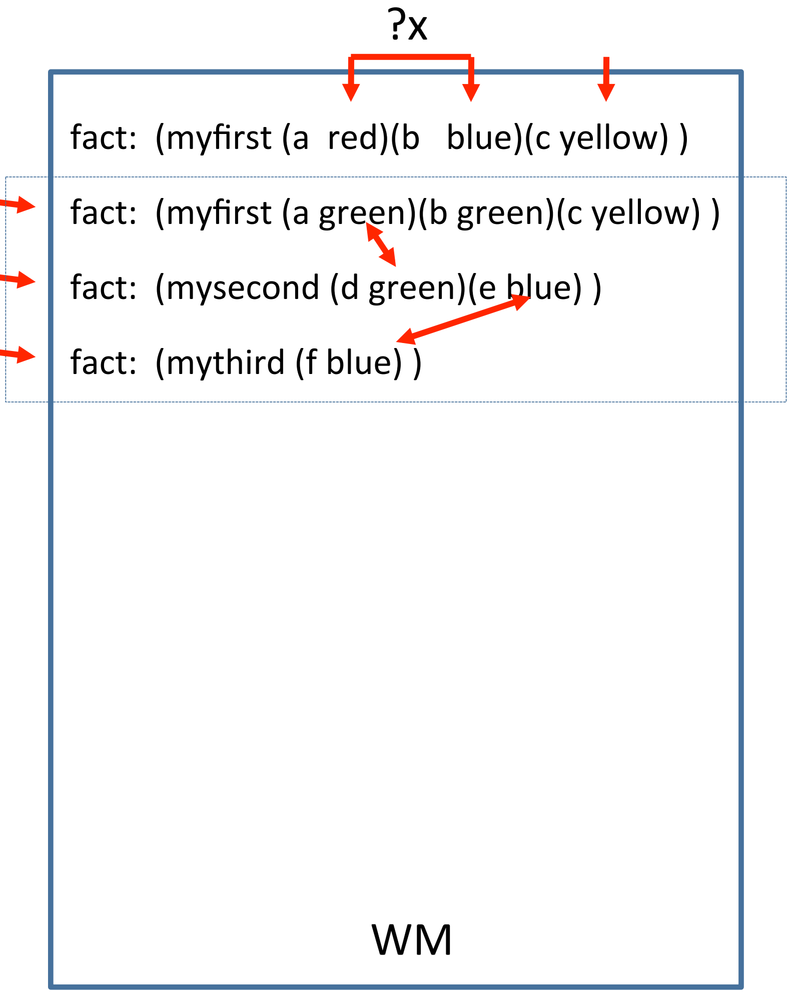


fact: (myfirst (a green)(b green)(c yellow))

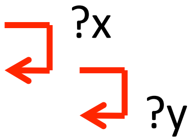
fact: (mysecond (d green)(e blue))

Example Rule-2


Pattern: (myfirst (a ?x))
Pattern: (mysecond (d ?x)(e ?y))
Pattern: (mythird (f ?y))
?x
?y


?x
fact: (myfirst (a red)(b blue)(c yellow))
fact: (myfirst (a green)(b green)(c yellow))
fact: (mysecond (d green)(e blue))
fact: (mythird (f blue))
WM

```
(defrule rule-2
    (myfirst    (a ?x))
    (mysecond   (d ?x)(e ?y))
    (mythird    (f ?y))
    =>
    (printout t "matched all "
                "of them" crlf)
)
```

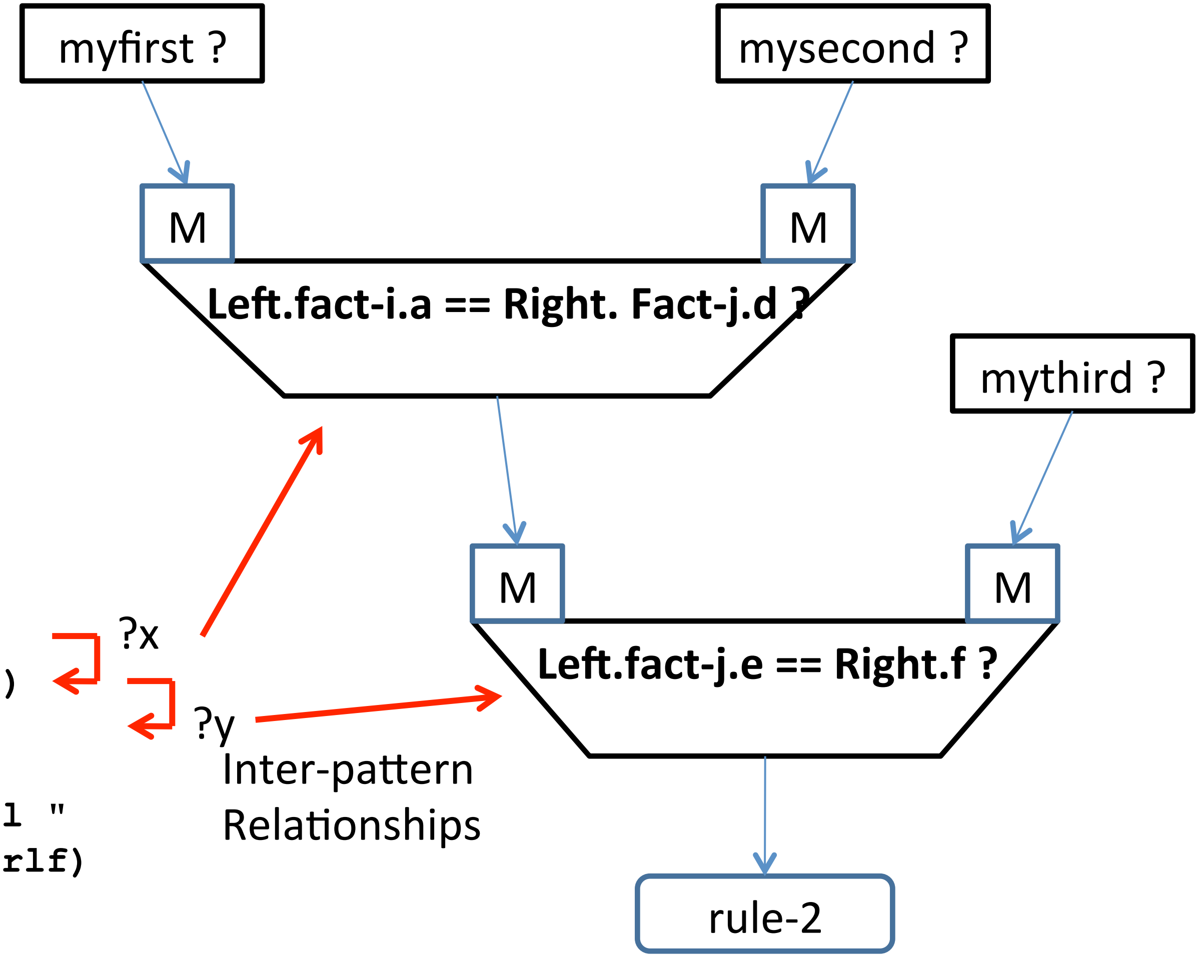


Compile Rule into a Rete Network

```
(deftemplate myfirst
                     (slot a)
                     (slot b)
                     (slot c)
(deftemplate mysecond
                     (slot d)
                     (slot e))
(deftemplate mythird
                     (slot f))
```

```
(defrule rule-2
     (myfirst   (a ?x))
     (mysecond  (d ?x)(e ?y))
     (mythird   (f ?y))
     =>
     (printout t "matched all "
                 "of them" crlf)
)
```

Rule rule-2



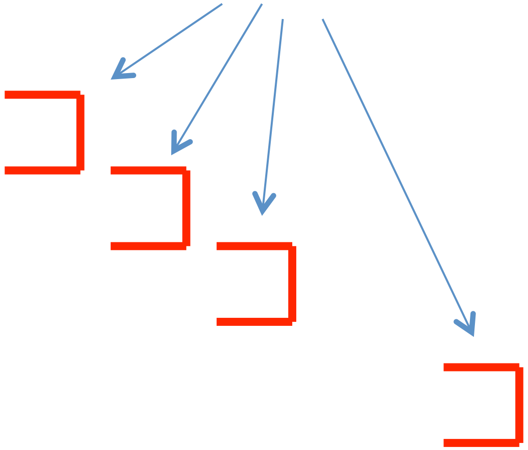
Rete Network

- A Rete network consists of
  - one-input nodes
  - two-input (“join” nodes),
  - Memory nodes and
  - terminal nodes
- The Rete network tests whether patterns match facts
  - One-input nodes test **intra-pattern** relationships due to re-occurrence of variables and constants
    - are created from a single pattern in a rule and test elements of single facts
  - Join nodes test **inter-pattern** relationships (relationships between patterns), due to occurring variables
    - Are created from two patterns in a rule

Possible Example Rule represented by a Network

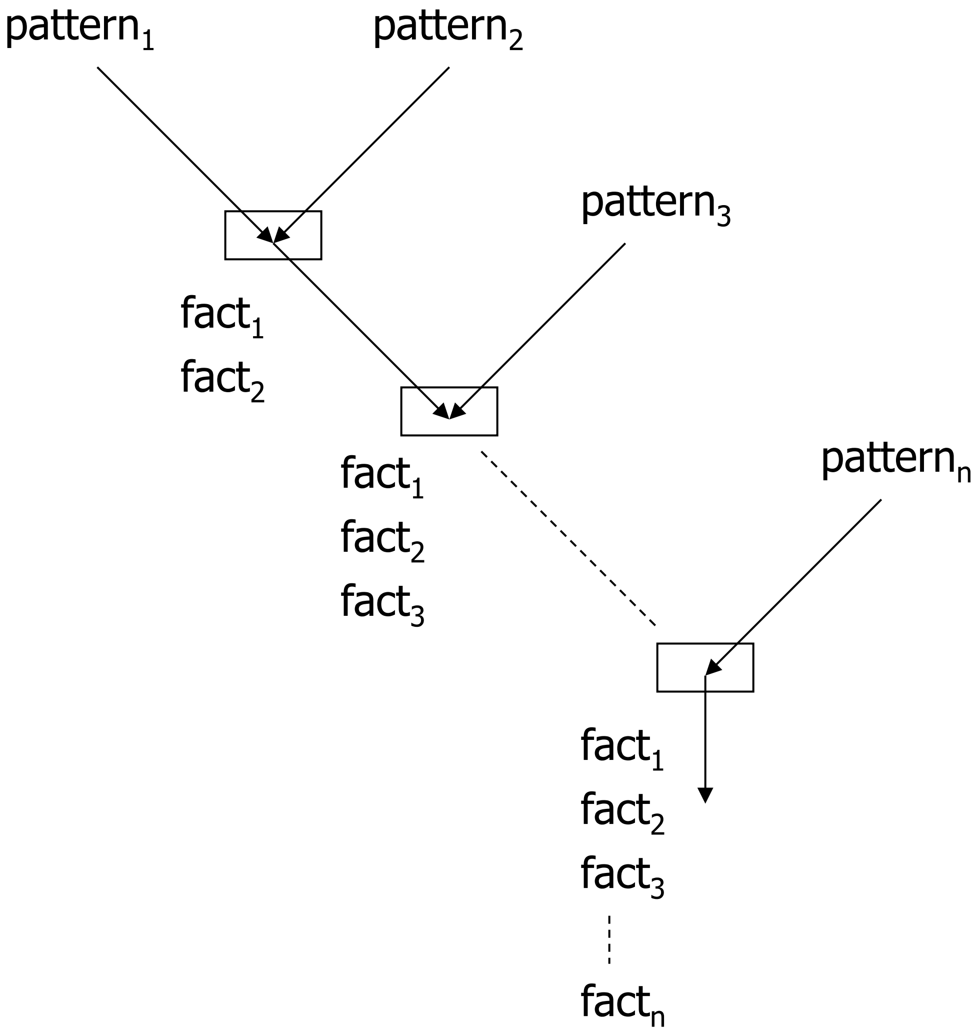
```
(defrule rule-n
     (myfirst    (a ?x)
     (mysecond   (b ?x)
     (mythird    (c ?x)
     .  .  .
     .  .  .
     (mynth      (n ?x)
     =>
     (printout t "matched all conditional elements" crlf)
)
```

Join Nodes connect pairs of patterns



Rete Network

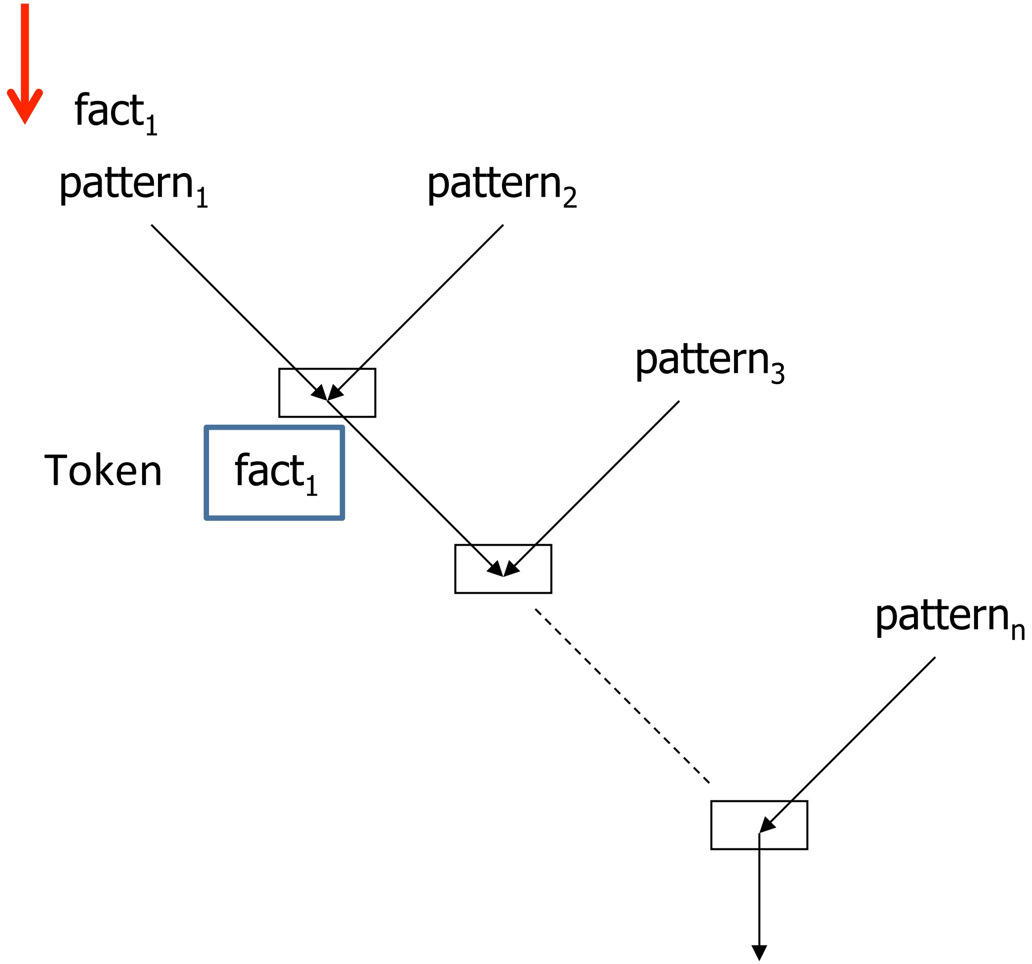
Organisation of the Working Memory



- RETE allows for efficient pattern matching
- All patterns found at the LHS of all rules are **compiled** into a network of connected patterns
- Partial matches are memorised in this network

Rete Network

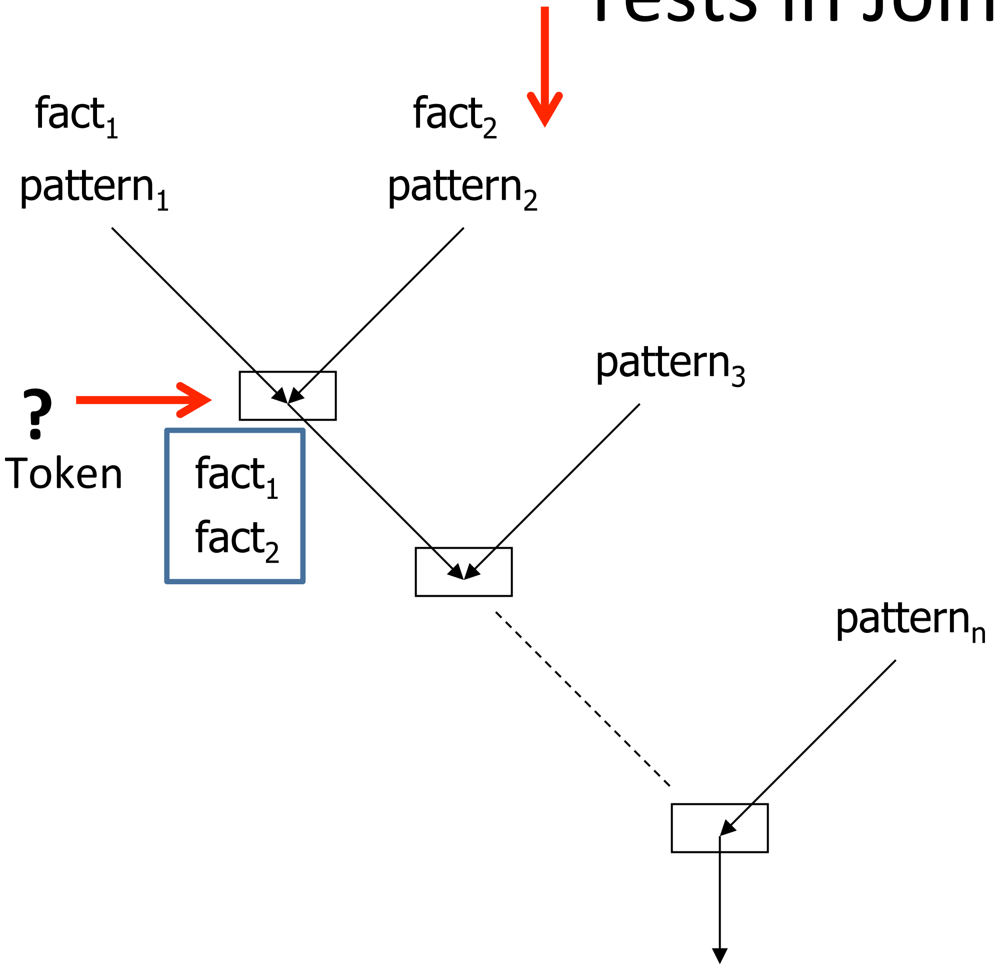
Adding Facts into Working Memory



- Patterns are like “**filters**” – they let facts through, if there is a match

Rete Network

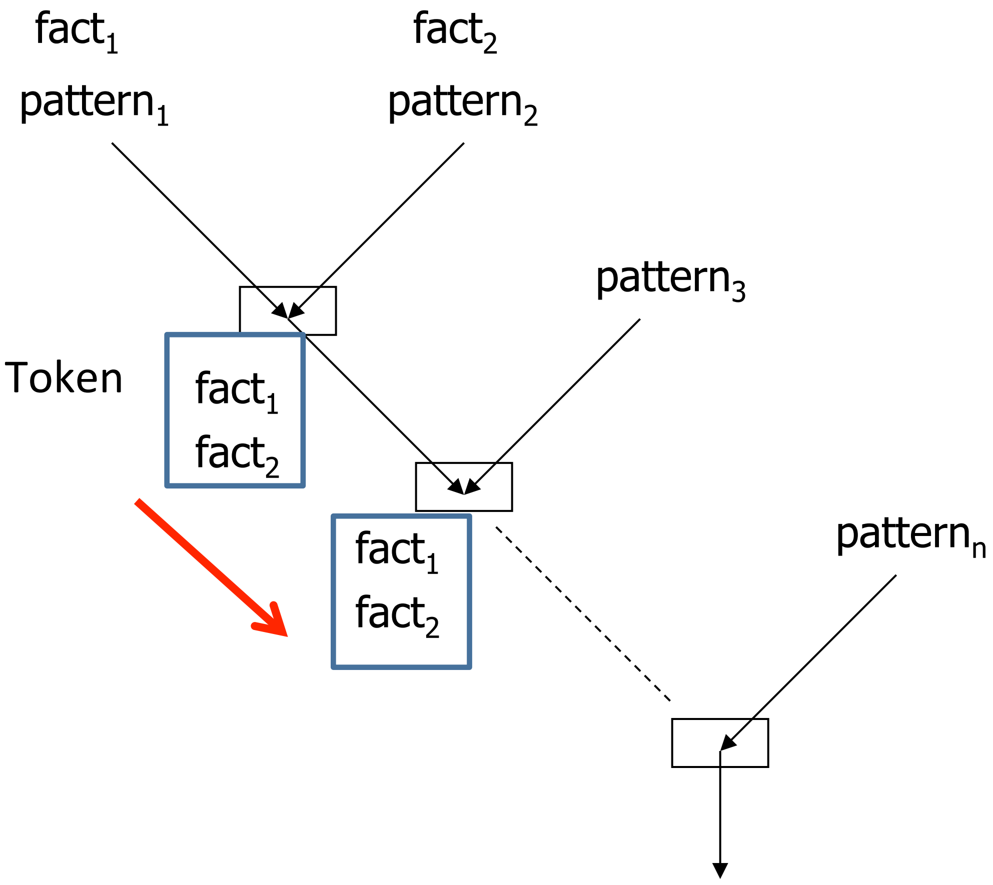
Add new Fact into Working Memory, execute Tests in Join Node



- A join node checks two patterns
  - Are some variables used in more than one pattern?
  - If yes, do they have the same value?
  - Are there any constraints specified over variables?
- All this is tested by a join node
- If the tests succeed, facts are sent to the next join node

Rete Network

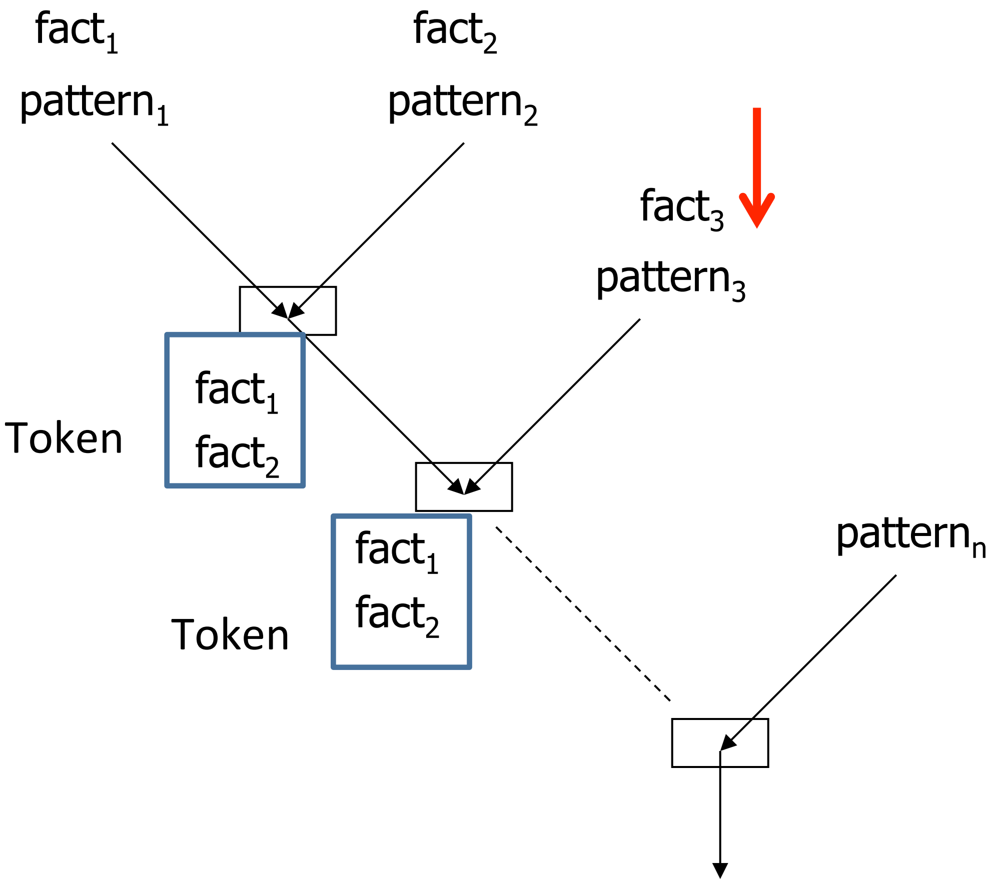
Propagating Facts to next Join Node



- If all the tests on a set of facts in Join nodes are OK, then this set of facts are propagated to the next Join node
- The set of facts, where Join node tests fail, remain at this Join node – they wait for the arrival of new facts

Rete Network

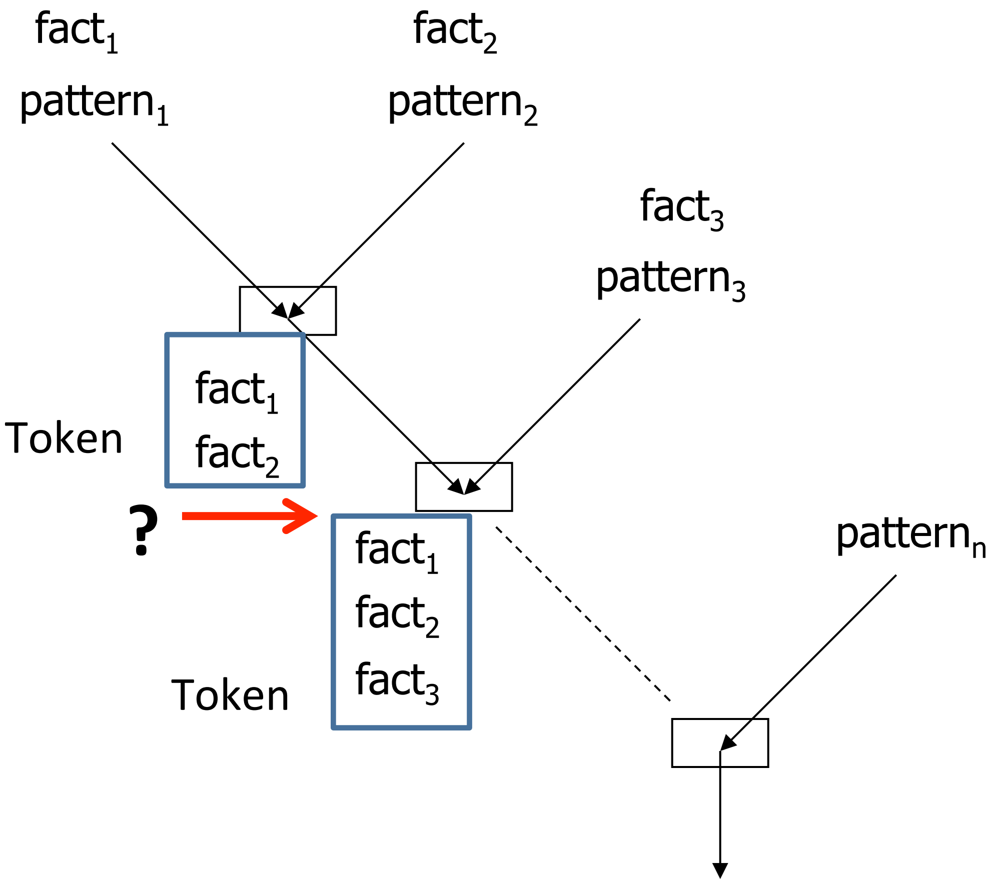
Adding new Fact, execute Tests in Join Node



- Adding a new fact for pattern 3 will lead to the execution of tests in the join node

Rete Network

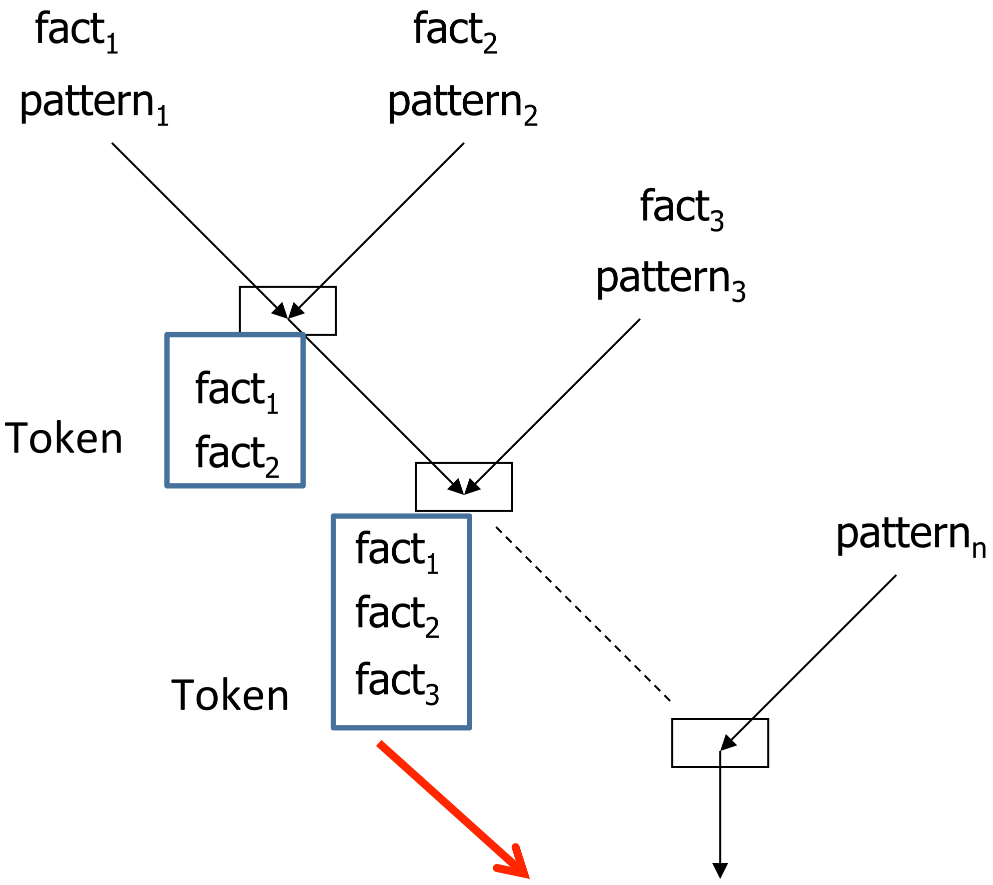
Adding new Fact, **execute Tests** in Join Node



- Adding a new fact for pattern 3 will lead to the execution of tests in the join node

Rete Network

Adding new Fact, **execute Tests** in Join Node

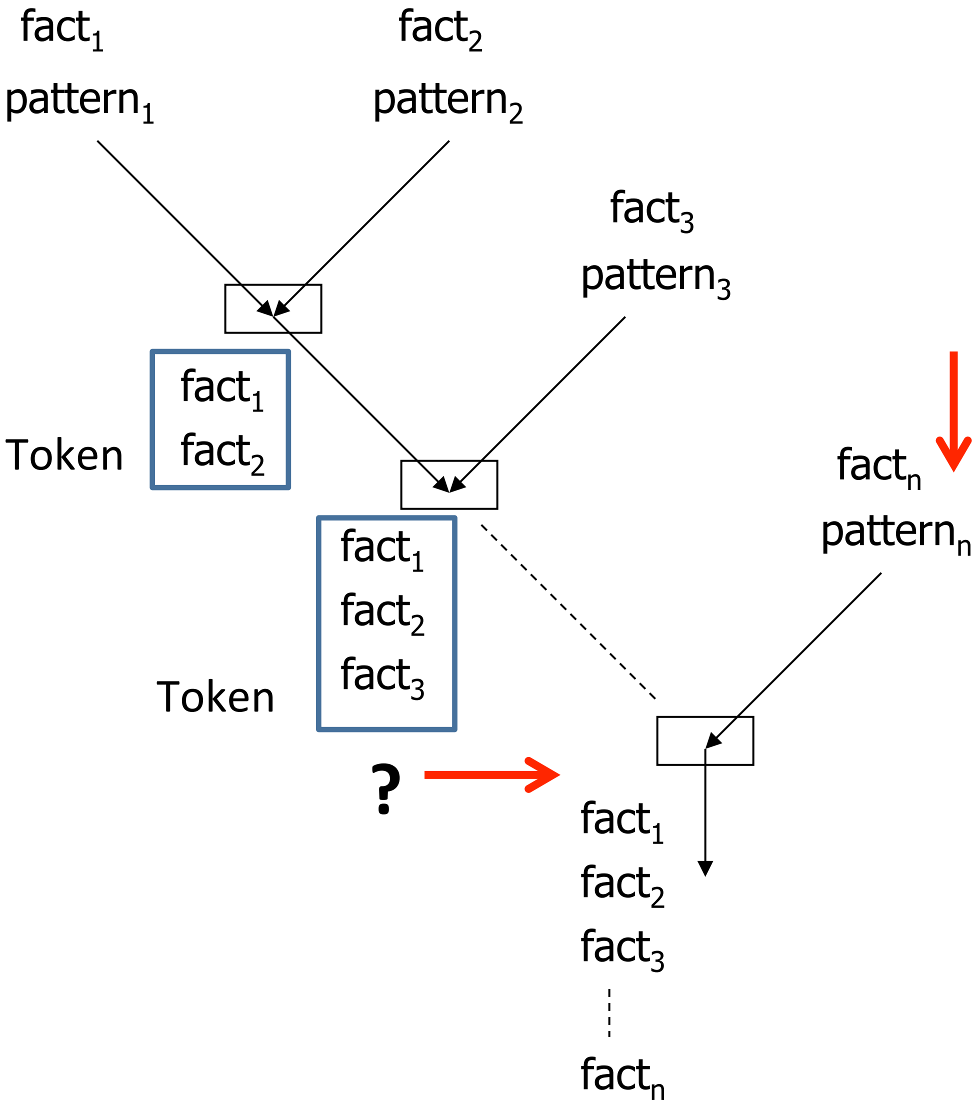


- Adding a new fact for pattern 3 will lead to the execution of tests in the join node

Question:
if pattern-3 is not matched, then what?

Rete Network

Adding Facts into Working Memory



- Finally, if a fact arrives for the last pattern of a rule and test in Join Node are OK, then the rule is **activated**! (written to the agenda)

Rete – A Network of Tests on Facts

- One and two-input nodes take facts from their inputs, apply tests and, if successful, send facts to their output
- Successful **intermediate matching results** are stored in memory nodes
- When all tests in join nodes are successful
  - We have a set of facts at the last output – these give us the bindings for variables in our rule
  - The rule is activated and written to the agenda

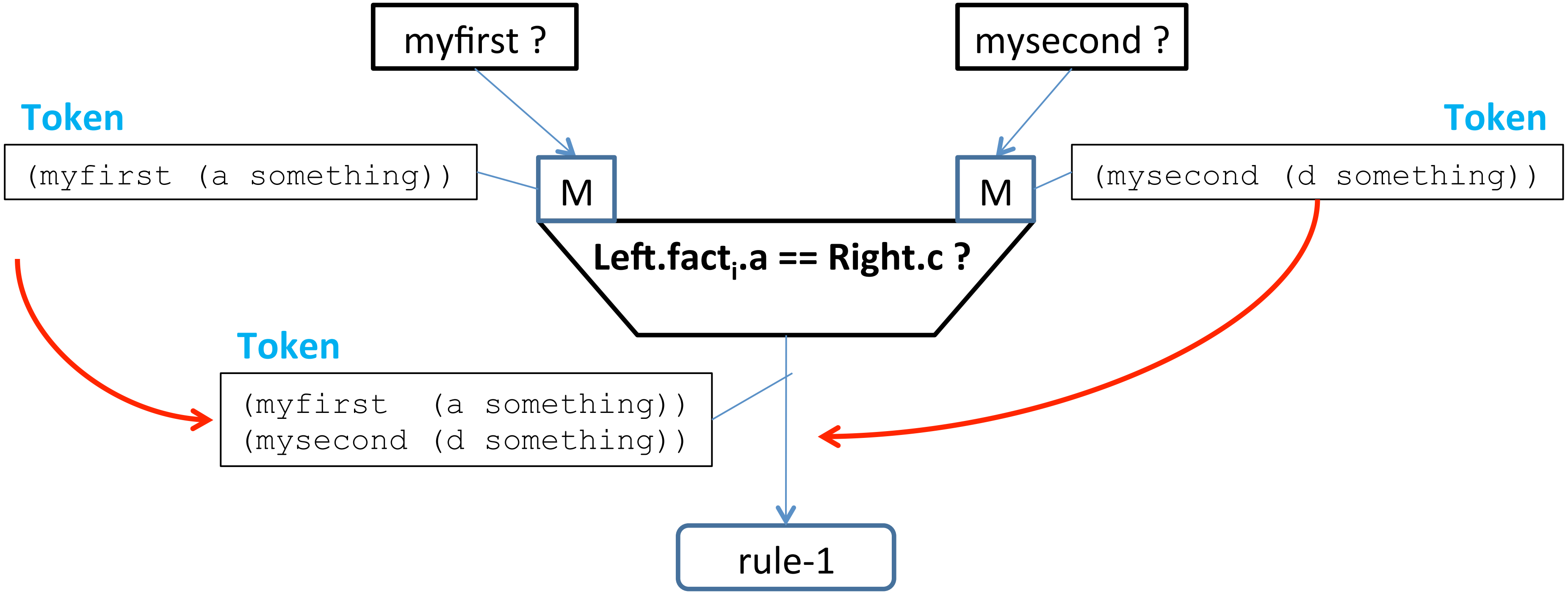
Asserting Facts into A Rete Network

- A fact “travels” from one test to the next, until a test fails
- The Rete network contains memory nodes, where facts are held that passed all tests up to this particular memory node
- Facts are transported by so-called “**tokens**”
  - Tokens are tagged as “ADD”, “DEL” etc.
- Join nodes have a left and right input memory where they store tokens
- Tokens can arrive at a join (two-input) node via its “left” or its “right” input
  - If two facts pass a test at a two-input node, then the fact from the right input is added to a copy of the token at the left input and this token is propagated further in the Rete network
  - **A token holds a list of facts** – tokens arriving at the left input of a two-input node grow with each successful test

Assert Facts into the Rete Network

```
(assert (myfirst  (a something)))
(assert (mysecond (d something)))
```

?x Inter-pattern / fact Relationship



myfirst ?

mysecond ?

Token

```
(myfirst (a something))
```

Token

```
(mysecond (d something))
```

M

M

Left.fact$_i$.a == Right.c ?

Token

```
(myfirst  (a something))
(mysecond (d something))
```

rule-1

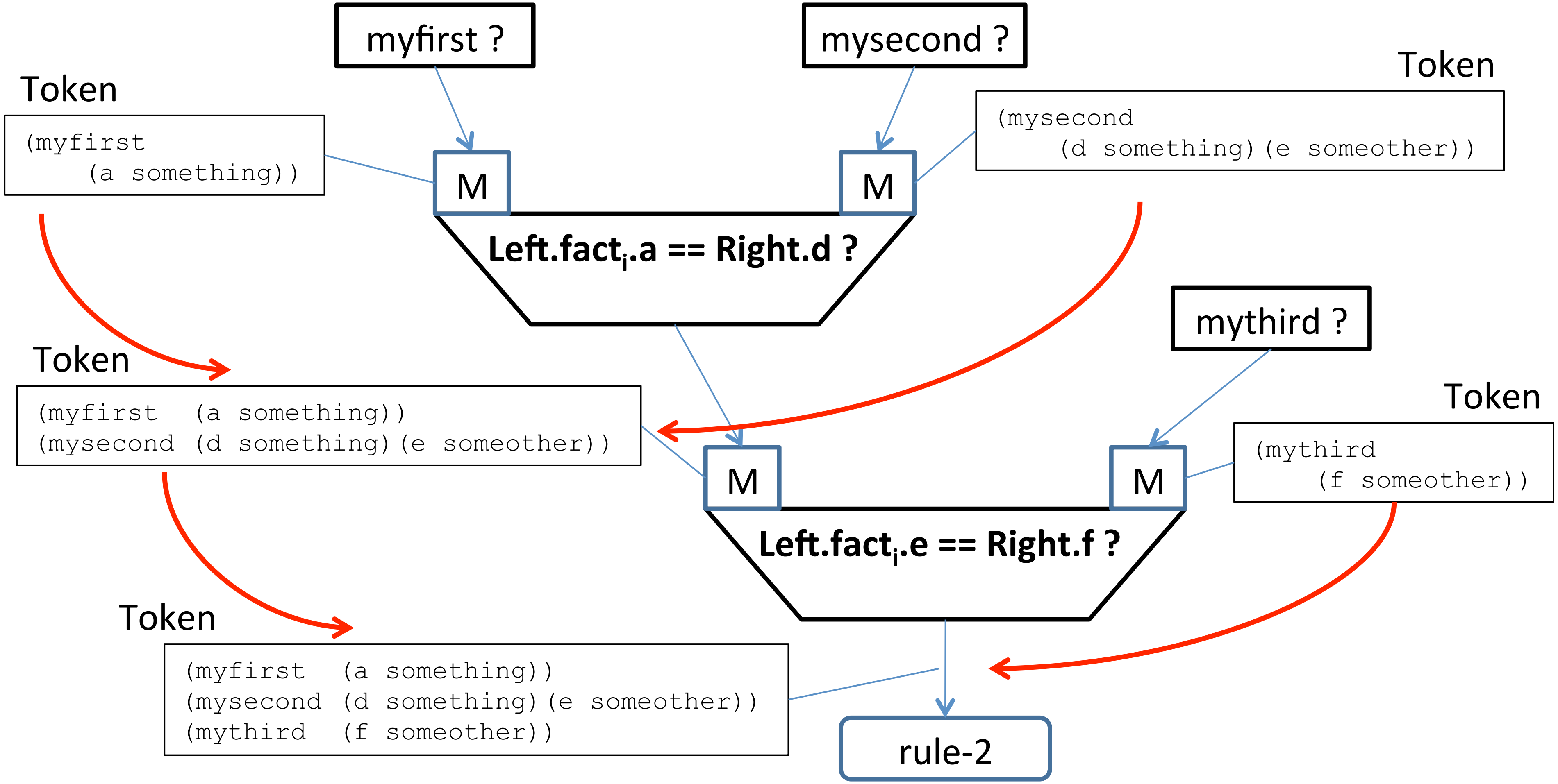
Assert Facts into the Rete Network

```
(assert (myfirst  (a something)))
(assert (mysecond (d something)(e someother))
(assert (mythird  (f someother)))
```

?x

?y

Inter-pattern / fact Relationships



```
(myfirst
    (a something))
```

```
(mysecond
    (d something)(e someother))
```

```
(myfirst  (a something))
(mysecond (d something)(e someother))
```

```
(mythird
    (f someother))
```

```
(myfirst  (a something))
(mysecond (d something)(e someother))
(mythird  (f someother))
```

Summary of Rete

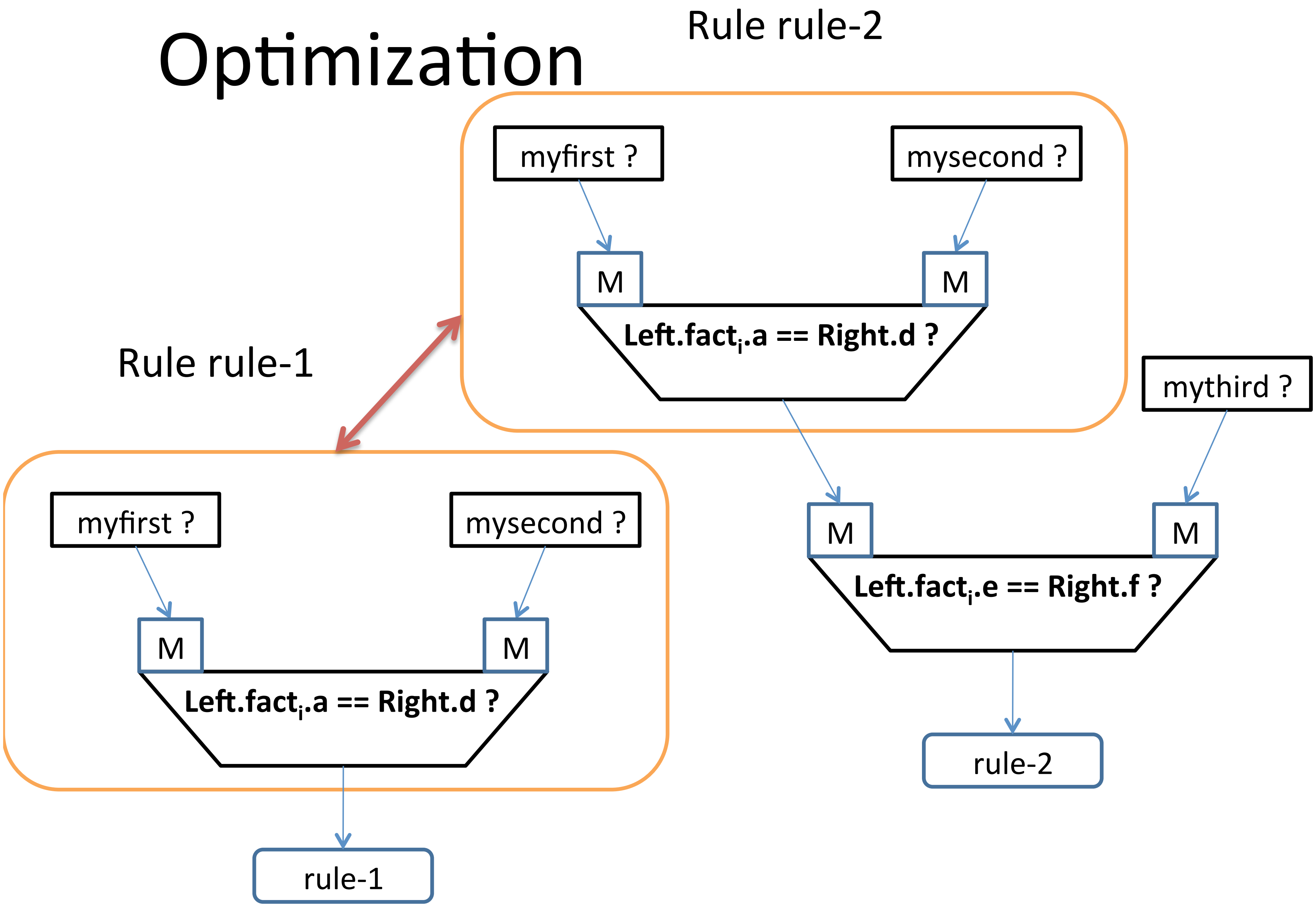
- A **Rete network** is a network of tests applied to facts asserted into Working Memory
- It stores **intermediate (partial) matching results** to save time in matching patterns against asserted facts
- Asserted Facts are transported with so-called "**tokens**" through the network
- **Join nodes** have a left and a right input memory
- Join nodes test facts arriving at their left or right input:
  - When a token arrives at one of these inputs, the facts transported by the token will be compared to facts held in the other memory
  - All successfully matched facts are collected into a new token and sent to the output of the join node
- Tokens arriving at the left input of a join node may transport more than one fact, whereas tokens arriving at the right memory only contain one fact

Optimization: Reuse parts of the Rete Network

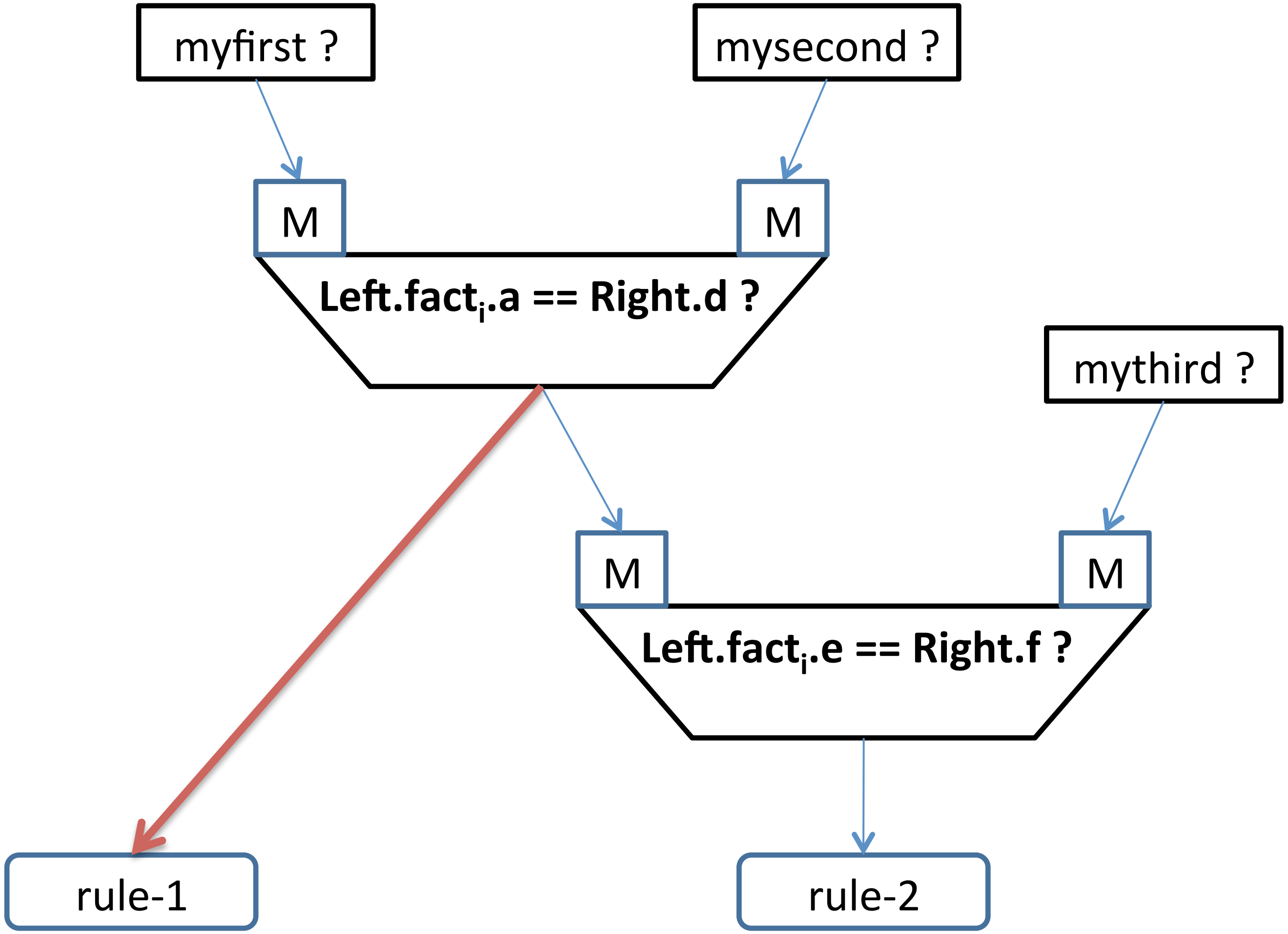
```
(deftemplate myfirst
               (slot a))
(deftemplate mysecond
               (slot b)
               (slot c))
(deftemplate mythird
               (slot d))

(defrule example-1
    (myfirst    (a ?x))
    (mysecond   (b ?x))
    =>
    (printout t "matched first "
                "and second" crlf)
)
(defrule example-2
    (myfirst    (a ?x))
    (mysecond   (b ?x)(c ?y))
    (mythird    (d ?y))
    =>
    (printout t "matched all "
                "of them" crlf)
)
```

- Observation
  - Both rules have **overlapping** lists of patterns in their LHSs:
    - Both rules have the same first and second pattern
    - The same inter-pattern test for ?x is needed for both rules
    - The Rete networks produced have strong similarities
- If Jess discovers these similarities during compilation of rules, it re-uses existing Rete network parts

Optimization

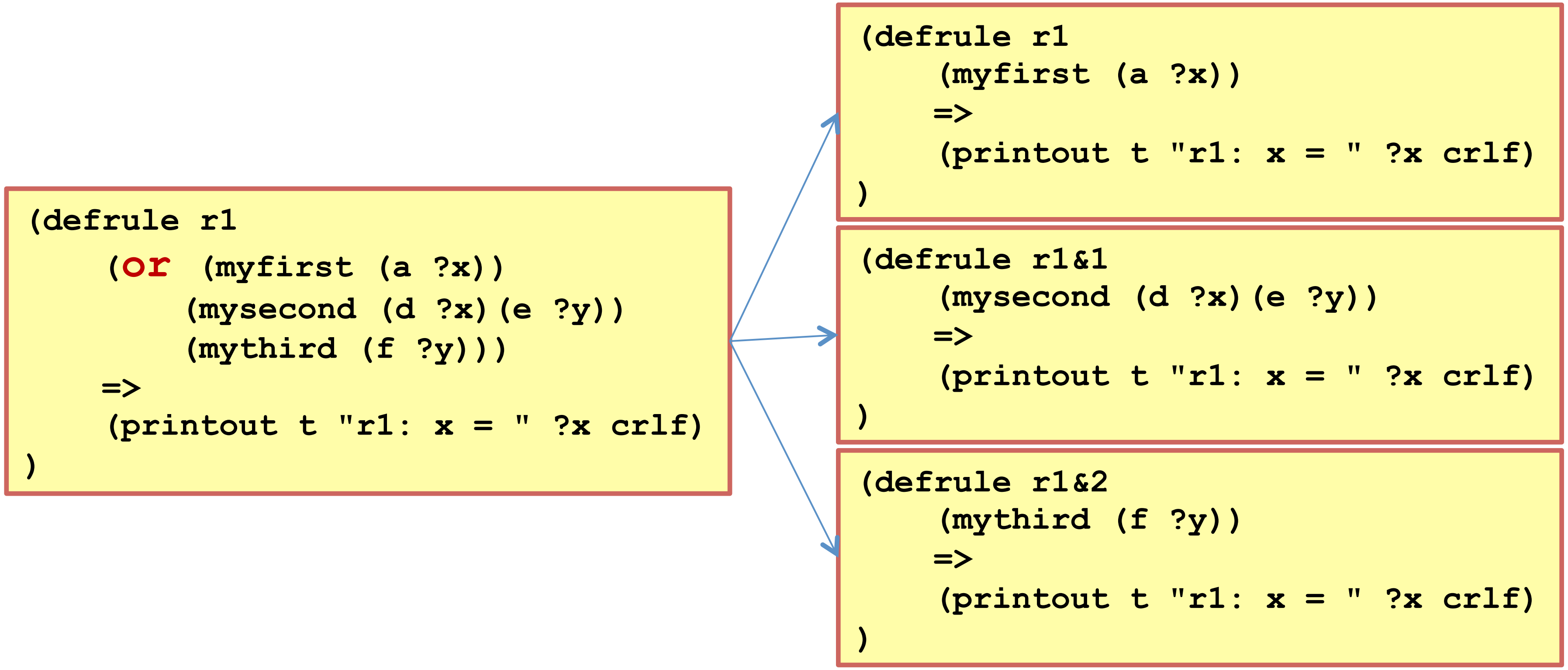


Optimization



Handling “or”

- A rule containing an **“or” conditional** element at its LHS with n patterns is equivalent to n rules with a LHS containing one of these patterns:
  - Jess creates “**subrules**” for each pattern in an OR conditional element



```
(defrule r1
    (or  (myfirst (a ?x))
         (mysecond (d ?x)(e ?y))
         (mythird (f ?y)))
    =>
    (printout t "r1: x = " ?x crlf)
)
```

```
(defrule r1
    (myfirst (a ?x))
    =>
    (printout t "r1: x = " ?x crlf)
)
```

```
(defrule r1&1
    (mysecond (d ?x)(e ?y))
    =>
    (printout t "r1: x = " ?x crlf)
)
```

```
(defrule r1&2
    (mythird (f ?y))
    =>
    (printout t "r1: x = " ?x crlf)
)
```

Handling “or”

- Each subrule is added separately to the Rete network
- Supporting node sharing in a Rete network for efficiency:
  - Similarities between LHSs of two rules can be used to share nodes in the Rete network between rules
  - If an OR conditional element is the first element of a LHS then no Rete network nodes can be shared between the subrules
    - Sharing only occurs as far as two rules’ LHSs are similar reading them from the top
  - Therefore:
    - Try to move OR conditional elements to the bottom of a LHS (if your design of a rule allows that)

Negation

- Careful with **negation**:

```
(defrule r1
    (myfirst (a ?x))
    (not(mysecond (d ?x)))
    =>
    (printout t "r1: x = " ?x crlf)
)
```

?

```
(defrule r2
    (not(mysecond (d ?x)))
    (myfirst (a ?x))
    =>
    (printout t "r2: x = " ?x crlf)
)
```

- A NOT conditional element tests the "absence of a fact
  - Rule r1: the absence of all those facts "`(mysecond (d ?x))`" is tested where ?x has a specific value
  - Rule r2: the absence of any fact "`(mysecond (d ?x))`" is tested – if one is present, the rule will not be activated

Handling Negation

- A **NOT conditional element** tests the **absence** of a fact
  - Therefore: it cannot provide bindings for variables in subsequent patterns of a LHS
- Evaluation of a NOT conditional element takes place:
  - When a matching fact is asserted – the pattern match “fails”
  - When a matching fact is removed – the pattern match “is successful”
  - When the pattern immediately preceding the NOT conditional element is evaluated

Handling Negation

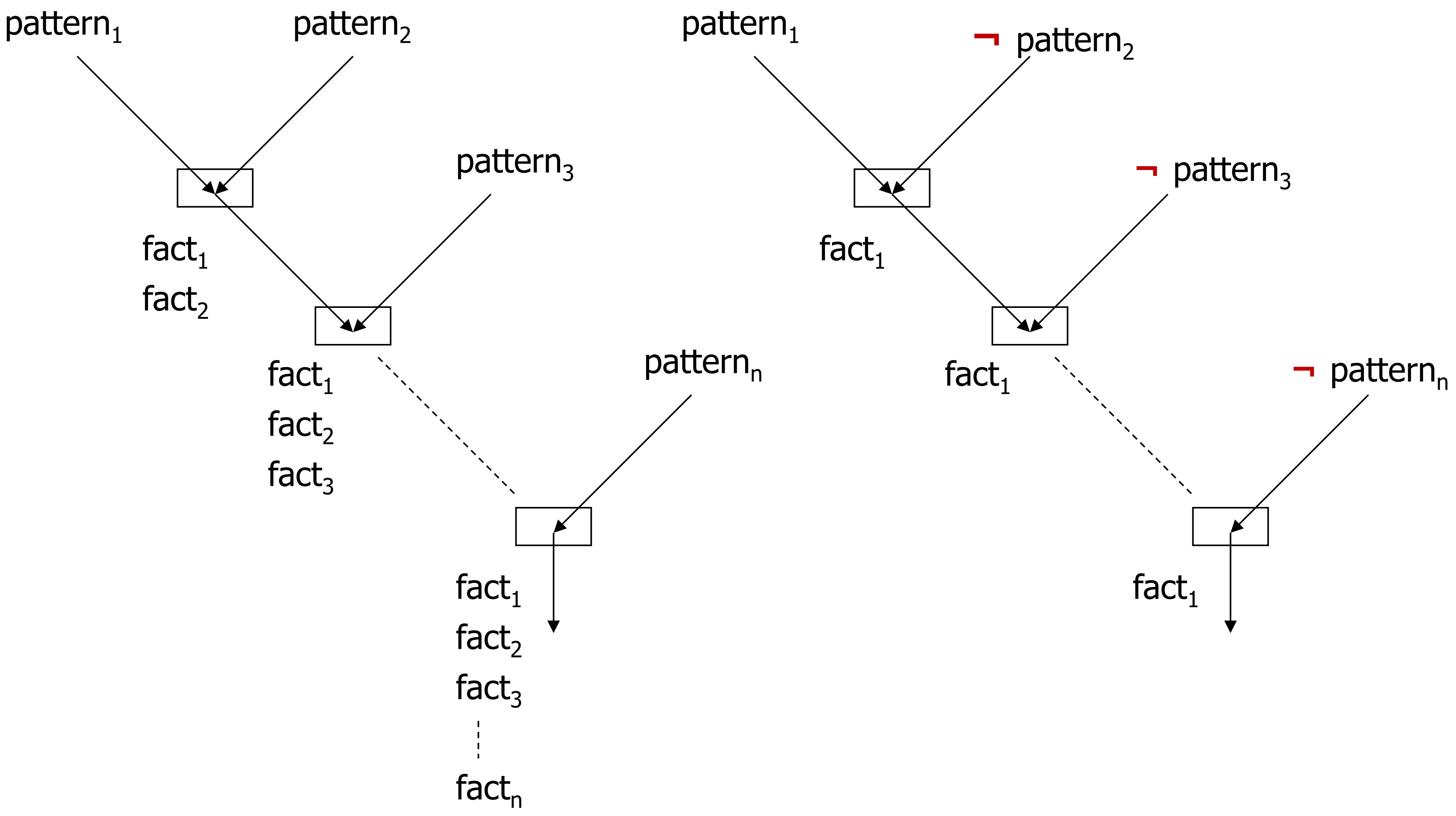
- If a NOT conditional element is the first pattern of a LHS:

```
(defrule r2
     (not(mysecond (d ?x)))
     (myfirst (a ?x))
    =>
     (printout t "r2: x = " ?x crlf)
)
```

```
(defrule r2
     (initial-fact)
     (not(mysecond (d ?x)))
     (myfirst (a ?x))
    =>
     (printout t "r2: x = " ?x crlf)
)
```

- Jess inserts `(initial-fact)` as the "immediate preceding pattern" in order to force an evaluation of a NOT conditional element

Rete Network Comparison



pattern$_1$

pattern$_2$

pattern$_3$

fact$_1$
fact$_2$

fact$_1$
fact$_2$
fact$_3$

pattern$_n$

fact$_1$
fact$_2$
fact$_3$
fact$_n$

pattern$_1$

¬ pattern$_2$

¬ pattern$_3$

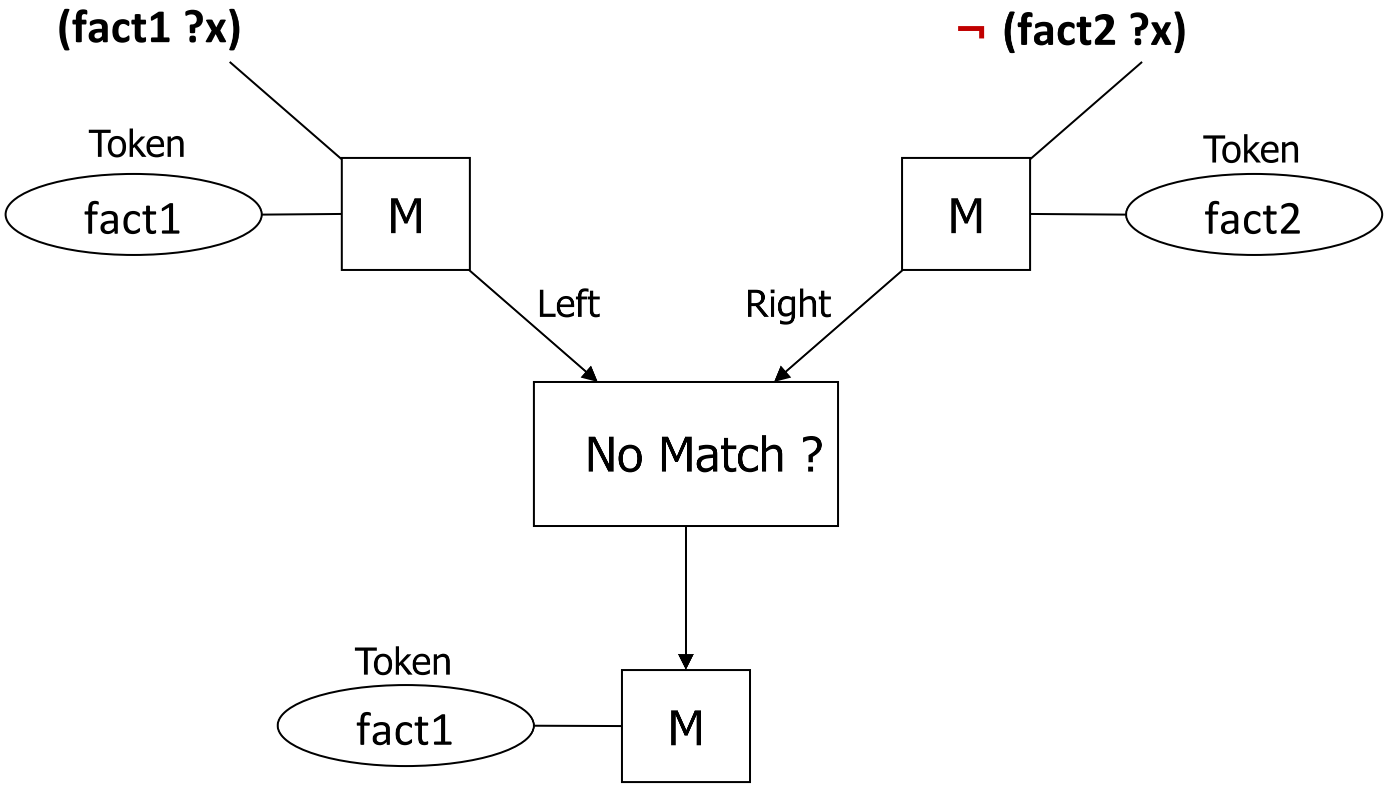
fact$_1$

fact$_1$

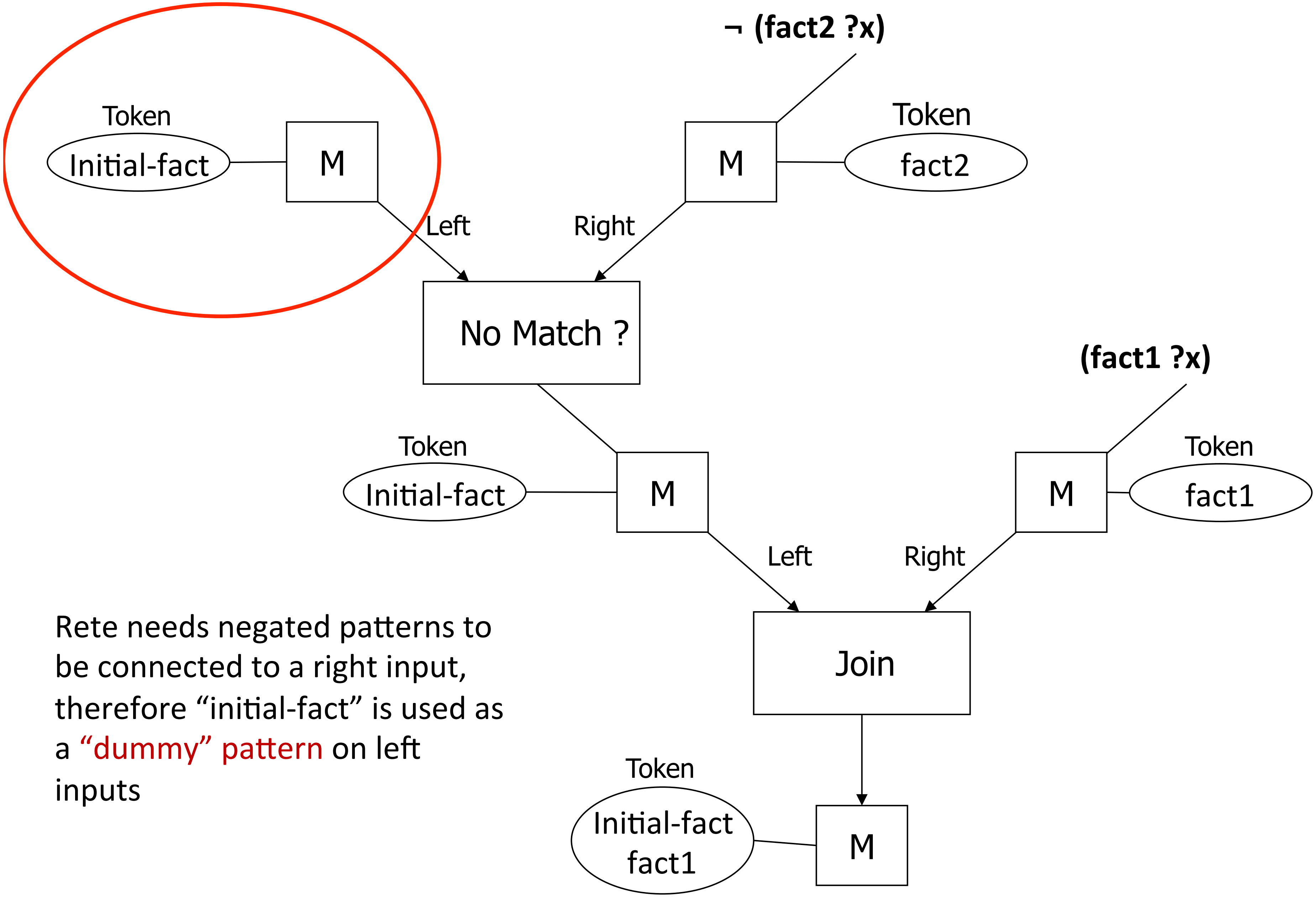
¬ pattern$_n$

fact$_1$

Rete Network with Negation



“initial-fact” for Negation



Rete needs negated patterns to be connected to a right input, therefore “initial-fact” is used as a “dummy” pattern on left inputs

Exploring the Rete Network

- Diagnostics during compilation
  - (watch compilations)
- Graphical representation of the Rete network
  - (view)
- Watch the content of left/right memory of join nodes used by rules
  - (matches <rule name>)

Constraints over Variables

- Constraints define restrictions on variables
  - Determine the range of values that can be bound to variables – influences how a pattern matches facts
  - We use the operators &, |, ~ for variable constraints
  - Simple constraints:
    - This pattern matches only those facts where slot c == “yellow”
      - `(myfirst   (a ?x)(b ?x)(c yellow))`
      - `(myfirst   (a ?x)(b ?x)(c ?y & yellow))`
    - This pattern matches only those facts where slot c == “yellow” or “green”
      - `(myfirst   (a ?x)(b ?x)(c yellow | green))`
      - `(myfirst   (a ?x)(b ?x)(c ?y & yellow | green))`
    - This pattern matches only those facts where c != “yellow”
      - `(myfirst   (a ?x)(b ?x)(c ~yellow))`
      - `(myfirst   (a ?x)(b ?x)(c ?y & ~yellow))`
    - This pattern matches only those facts where c != “yellow” and c != “green”
      - `(myfirst   (a ?x)(b ?x)(c ~yellow & ~green))`
      - `(myfirst   (a ?x)(b ?x)(c ?y & ~yellow & ~green))`

What is the difference?

What is the difference?

Constraints over Variables

- Constraints define restrictions on variables
  - More complex constraints
    - This pattern matches only those facts where slot c != slot b
      - `(myfirst   (a ?x)(b ?x)(c ?y & ~?x))`
      - `(myfirst   (a ?x)(b ?x)(c ?y & :(neq ?y ?x)))`
    - This pattern matches only those facts where slot b > slot a
      - `(mythird   (a ?x)(b ?y & :(> ?y ?x)))`
    - This pattern matches only those facts where the value of slot b is equal to the result of the return value of a function
      - `(mythird   (a ?x)(b ?y & :(myFunction ?x)))`

Summary

- Jess Efficiency

  …
- Question?