APLAI
Rule-based systems: Jess

Gerda Janssens
Departement computerwetenschappen
A01.26
http://www.cs.kuleuven.be/~gerda/APLAI

Topics

- Rule-based systems
- Jess programming
- Rete

Jess

- Jess user's mailing list
- Jess in Action, Rule-Based Systems in Java, Ernest Friedman-Hill, Manning, 2003.
  ISBN 1-930110-89-8
  Plaatsingsnummer: **6 681.3*D32 JAVA/2003**
  a hands-on introduction to rule-based programming using Jess.

Rules

- A rule is a kind of instruction that applies in certain situations.
- No smoking in restaurants!
- IF I am smoking a cigarette

  AND I am entering a restaurant

  THEN throw away the cigarette
- LHS, premises, conditions
- RHS, conclusions, actions

Rule-based systems

- Systems that uses rules to derive conclusions from premises.
- Rule engines are part of a rule development and deployment environment.
- Declarative programming!!!
- Business rules: nowadays
- Expert systems: success story for AI in 1970's and 1980s.
  - Medical diagnosis (MYCIN), engineering, chemistry, computer sales (XCON for DEC)

Current rule-based systems (RBSs)

- Focus on a specific problem domain.
- Are routine now: ubiquitous!!!
- Efforts to standardize rule engine APIs etc…
- Either replace human expertise or automate and codify business practices.
- Advise salespeople, financiers, medical people ,…
- To order supplies, monitor industrial processes, route telephone calls, process web forms, to filter emails.

Current business rules systems

- http://www.businessrulesgroup.org
  The Business Rules Manifesto
- Haley company: also natural language interface
  (bought by Oracle, October 2008)
- http://www.exsys.com/index.html
  knowledge automation expert systems with focus on solving real-world problems through a practical easy-to-use development paradigm; has some demo's on their site.
- ILOG (acquired by IBM, July 2008), XpertRule

JESS programming

The Java Expert System Shell = Jess

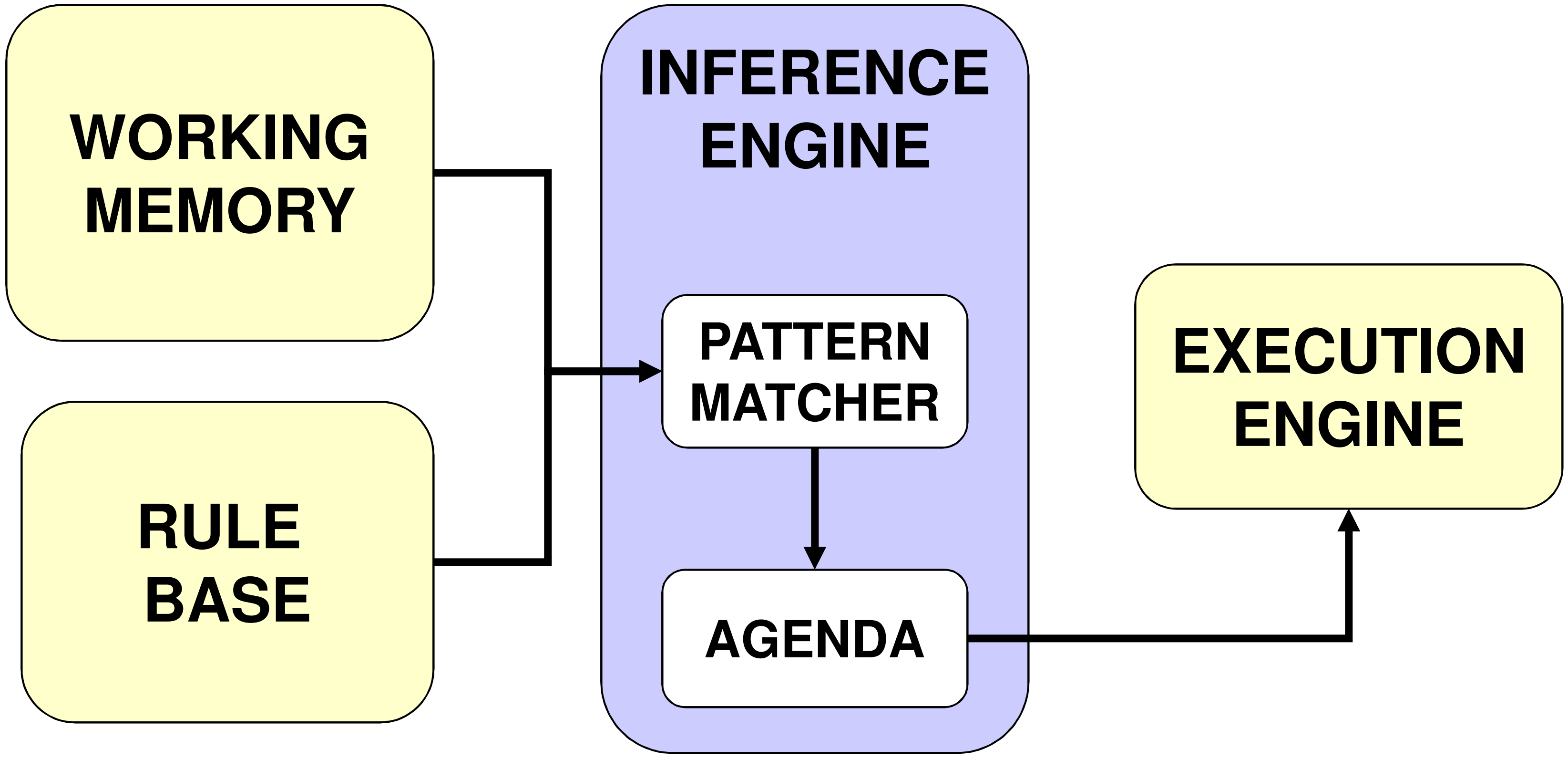
- www.jessrules.com
- Developed at Sandia National Laboratories in late 1990s.
- Created by Dr. Ernest J. Friedman-Hill.
- Inspired by the AI production rule language **CLIPS** (NASA, early 80s).
- Fully developed Java API for creating rule-based expert systems.

More examples

- In the Jess book complete examples of
  - Tax forms advisor
  - Diagnostic assistant
  - Fuzzy logic controller
  - Web agent
  - J2EE apps
- CLIPS site (also object-oriented style)

http://clipsrules.sourceforge.net/

Jess architecture diagram



Architecture of a rule-based system

- A rule base
- A working memory
  - The specific data: the fact base
- An inference engine
  - A pattern matcher: comparing all the rules to the working memory to decide which rules should be activated.
  - An agenda: list of activated rules = conflict set
  - An execution engine: fires the selected rule.

How does Jess work?

- Jess matches **facts** in the fact base to **rules** in the rule base.
- The **rules** contain function calls that manipulate the fact base and/or other Java code.
- Jess uses the **Rete** (ree-tee) **algorithm** to match patterns.
- **Rete network** = an interconnected collection of nodes = working memory.

First Jess program

```
(printout t “Hello APLAI-ers!” crlf)
```

To load the program `(batch “hello.clp”)`

Lists in Jess

```
(a b c)              ; list of tokens
(1 2 3)              ; list of integers
(+ 2 3)              ; an expression
(“Hello world!”)     ; a string
(foo ?x ?y)          ; a function call
```

Jess variables

- Named containers that hold a single value.
- Untyped. Begin with a ? mark.
- Can change types during lifetime.
- Assigned using **bind** function.

```
(bind ?x 2)                    ; assign x = 2
(bind ?y 3)                    ; assign y = 3
(bind ?result (+ ?x ?y))       ; find sum
```

Also functions are lists

```
(deffunction get-input()
“Get user input from console.”
(bind ?s (read))
(return ?s))

(deffunction area-sphere (?radius)
“Calculate the area of a sphere”
(bind ?area (* (* (pi) 4)(* ?radius ?radius)))
(return ?area))
```

call to calculate the area of sphere 2 meters in radius?

```
(printout t
   "The surface area of a radius = 2 meter sphere is “
     +  (area-sphere 2) + " m^2")
```

Control Flow

- Control flow
  - (foreach <var> <list> <expression>+)
  - (if <boolean expr> then <expr>+ [else <expr>+])
  - (while <boolean expr> do <expression>+)
- Jess specific
  - (apply (read) 1 2 3)
  - build – eval: convert data to code (eval "(+ 1 2 3)")
  - (progn <expr>+) evaluates the exprs,
    returns the value of the last one

Common functions

- (reset): resets the Working Memory
- (clear): clears Jess (more deeply than reset)
- (facts): shows facts currently in WM
- (rules): shows rules currently in WM
- (batch <filename>): parse and evaluate the give file as Jess code
- (run): starts the engine
- (apply) calls a function on a given set of arguments.
- (build) parses and executes a function from a string.
- (eval) “”
- (open <filename> id w): opens a file in write mode
- (read t): reads from Standard Input
- (read id): reads the file content
- (printout t “…” crlf): prints to StdOutput
- (printout id “…” crlf): prints to file
- (exit): exits the Shell

- Jess comes with 200 built-in functions and you can define functions.
- This makes Jess a full-featured programming language where you can write any program
- You can call ANY Java function from within Jess!!! (link to databases, GUI forms … full-fledged applications)
- % access public instance/class variables

```
Jess> (bind ?pt (new java.awt.Point))
<Java-Object:java.awt.Point>
Jess> (set-member ?pt x 45)
45
Jess> (set-member ?pt y 89)
89
Jess> (get-member ?pt x)
45
```

Creating Java objects

```
Jess> (import java.util.*)
Jess> (bind ?prices (new HashMap))
<external-address:java.util.HashMap>

%calling the method put
Jess> (call ?prices put bread 1.80)
Jess> (?prices put peas 1.99)

Jess> (?prices get bread)
1.80
```

Jess rule-based programming

- Data
- Rules
- Inference engine

- Focus on Jess (skip the Jess-Java slides)

The facts in the knowledge base

- Ordered facts are simply lists, where the first field acts as a sort of category for the fact.

```
(shopping-list eggs milk bread)
(person  “Ernest Friedmann” Male 18)
(father-of jan piet)
```

- Facts have a **head** and one or more **slots**.
- Slots hold data.

Example

```
Jess Version 7.0 11/1/2006

Jess> (reset)    ; initializes the WM
TRUE
Jess> (assert (father-of jan piet))
<Fact-1>

Jess> (facts)
f-0   (MAIN::initial-fact)
f-1   (MAIN::father-of jan piet)
For a total of 2 facts in module MAIN.
Jess>
```

- More functions for retracting, modifying and duplicating facts

```
Jess> (watch facts)
TRUE
Jess> (reset)
 ==> f-0 (MAIN::initial-fact)   % fact being added
TRUE
Jess> (facts)
f-0   (MAIN::initial-fact)
For a total of 1 facts in module MAIN.
Jess> (assert (father-of j p ))
 ==> f-1 (MAIN::father-of j p)
<Fact-1>

Jess> (assert (father-of ?x k))
Jess reported an error in routine Context.getReturn
        while executing (assert (MAIN::father-of ?x k)).
  Message: No such variable x.
  Program text: ( assert ( father-of ?x k ) )  at line 16.
Jess>
```

Unordered facts

- More structure: use names for the fields in the facts.
- Cannot be asserted until you define their structure

  (deftemplate <template-name>
  ((slot | multislot) <slot-name>)*)

```
(deftemplate person  “Person in our DB”
  (slot name)
  (multislot gn)         ; any number of given_names
  (slot age  (default 21)))
```

Unordered facts

```
(deftemplate person  “Person in our DB”
(slot name)
(multislot gn)             ; given_names
(slot age  (default 21)))

 Jess> (assert (person (gn jan piet) (name smith)))
  ==> f-1 (MAIN::person (name smith) (gn jan piet)
    (age 21))
<Fact-1>

Jess> (assert (person (name smith)))
  ==> f-2 (MAIN::person (name smith) (gn ) (age 21))
<Fact-2>
```

Typical facts

- **Ordered** – head only.

```
;; An ordered fact with no slots - a placeholder
  that indicates state.
(assert (answer-is-valid))      ;; used as a trigger
```

- **Ordered** – single slot.

```
;; A ordered fact of one slot   a problem parameter
(assert (weightfactor 0.75))
```

- **Unordered** – multiple slot, like a database record.
- **Shadow** – slots correspond to properties of a JavaBean.

Put Java objects into working memory

- Shadow facts are unordered facts whose slots correspond to the properties of a JavaBean.
- With the JavaBeans API you can create reusable, platform-independent components. Using JavaBeans-compliant application builder tools, you can combine these components into applets, applications, or composite components.
- `defclass` – creates a deftemplate from a bean.
- `definstance` – adds bean to working memory.

An example JavaBean with 1 property

```
public class DimmerSwitch {
  private int brightness = 0;
  public int getBrightness() { return brightness; }
  public void setBrightness(int b) {
      brightness = b;
      adjustTriac(b);
  }
  private void adjustTriac(int brightness) {
      // Code not shown
  }
}
```

Java → Jess

```
Jess> (defclass dimmer DimmerSwitch)
```

This generate a deftemplate that represent the class

```
Jess> (bind ?ds (new DimmerSwitch))
```

This creates an instance (object) of DimmerSwitch

(it's a Java Object)

```
Jess> (definstance dimmer ?ds)
```

This adds the object into the working memory

```
Jess> (call ?ds setBrightness 10)
```

This is a call to a method of Dimmerswitch class

that changes the attribute of the object referenced by ?ds.

Java → Jess: PropertyChangeSupport

1) Add this code in the class definition

```
private PropertyChangeSupport pcs =
      new PropertyChangeSupport(this);
public void
addPropertyChangeListener(PropertyChangeListener p) {
   pcs.addPropertyChangeListener(p);
}
public void
removePropertyChangeListener(PropertyChangeListener p)
   {
   pcs.removePropertyChangeListener(p);
}
```

2) Change the Set Method of the JavaBean

Java → Jess: propertyChangeSupport

2) Change the Set Method of the JavaBean

Extra code in the setBrightness method of the example:

```
public void setBrightness(int b) {
       int old = brightness;
       brightness = b;
       adjustTriac(b);
       pcs.firePropertyChange(“brightness",
              new Integer(old), (new Integer(b)));
}
```

Jess

- Data
- Rules
- Inference engine

Writing Rules in Jess

- Given two ordered facts (a 1 3) and (a 2 3), a new fact (b 1 2 3) will be asserted

```
Jess> (defrule define-b-from-a
       (a 1 3)
       (a 2 3)
       => (assert ( b 1 2 3)))
TRUE
Jess> (facts)
f-0   (MAIN::a 1 3)
f-1   (MAIN::a 2 3)
For a total of 2 facts in module MAIN.
Jess> (run)
 ==> f-2 (MAIN::b 1 2 3)
1                  ; 1 rule fired
```

Jess help

```
Jess> (help reset)
Removes all facts from working memory, removes all activations, then
asserts the fact (initial-fact) , then asserts all facts found in
deffacts, asserts a fact representing each registered Java object,
and (if the set-reset-globals property is TRUE) initializes all
   defglobals.


Jess> (help run)
Starts the inference engine. If no argument is supplied, Jess will
keep running until no more activations remain or halt is called. If
an argument is supplied, it gives the maximum number of rules to fire
before stopping. The function returns the number of rules actually
fired.
Jess>
```

Wrong rule

- Left hand side of rule matches facts, not functions such as eq

-
  ```
  Jess> (defrule wrong-rule
                 (eq 1 1)
                   =>
                   (printout t “Just as I thougth!” crlf))
  ```

Conditional elements

- RHS of rules contain function calls
- Patterns in LHS of rule are combined with AND by default
- Conditional elements OR/NOT/AND can be used/nested

-
  ```
  (defrule or-not-and-ex
       ( or  (not (a))
              (and (b) (c))
        ) => ….)
  ```

Matching patterns

```
Jess> (defrule ex-1
           (a ?x ?y)
           => (printout t "seen a "  ?x crlf))
TRUE
Jess> (watch all)
TRUE
Jess> (a 1 2)
Jess reported an error in routine Funcall.execute
       while executing (a 1 2).
  Message: Undefined function a.
  Program text: ( a 1 2 )  at line 7.
```

Matching patterns

```
Jess> (defrule ex-1 (a ?x ?y) => (printout t "seen a
   "  ?x crlf))
TRUE
Jess> (watch all)
TRUE
Jess> (assert (a  1 2 ))
 ==> f-0 (MAIN::a 1 2)
==> Activation: MAIN::ex-1 :  f-0
<Fact-0>
Jess> (run)
FIRE 1 MAIN::ex-1 f-0
seen a 1
1
```

More matching patterns

```
(defrule ex-2
   (not-b-and-not-c ?n1&~b ?n2&~c)
   (client (city Leuven|Brussel))
   (different ?d1 ?d2&~?d1)
   (same ?s ?s)
   (more-than-one-hundred ?m&:(> ?m 100))  ;predicate fct
   =>
   (printout t “found!!” crlf))

(defrule ex-3                ; test is a conditional expression
   ?c <- (car (age ?x))          ; and is evaluated after
   (test (>  ?x 20))             ; the previous pattern
   =>
   (retract ?c))
```

Matching pattern for multislot

```
(deftemplate emp
      (slot id (type integer))
      (slot name )
      (multislot dept ))
```

To match any and all departments in the multislot, you need to use a multifield (a special kind of list), like "$?dp".

```
(emp (id ?id) (name ?fn) (dept $?dp))
```

Multifield

- Multifields are generally created using special multifield functions like create$ and can then be bound to multivariables:

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

(emp (id ?id) (name ?fn)
     (dept $?dp:(< (length $?dp) 3)))
```

Semantics of not

- (not (cond)) is fulfilled if there exist no facts that fulfill cond, i.e. “not exists”
- Variables used within “not” must not be referred to from outside the not.

```
(a ?x)
(not (and (b ?y)  (c ?y))) …
```

- There is an a-fact, but no b-fact with the same value as a c-fact

Be careful

```
(defrule no-red-cars
 (car  (color ~red))
 =>
)
; fires for each car that is not red
; no red cars??
(defrule no-red-cars-2
   (not (car (color red))) => )

; fired when a matching fact is asserted
;      when a matching fact is removed
;      when the pattern just before it is evaluated
;           can be (initial-fact)
```

Jess versus CHR /Prolog

```
(defrule find-c
   (a ?x ?z)
   (b ?z ?y)
   (test (< ?x 20))
   =>
   (assert (c ?x ?y)))

c(X,Y) :- a(X,Z), b(Z,Y), X < 20 .

a(X,Z),b(Z,Y) ==> X < 20 | c(X,Y).
% what with simplification???
```

Some Exercises: Ex 1

-
  ```
  (person-has jeff car)
  (person-has jeff ipod)
  ….
  ```
- ;; when a person has 3 objects, then ….
  ```
  (defrule person-has-3-objects
  ```

Ex 2

- ;; when a person has a car or an ipod, then
  …
- ;; when the temperature is between 1 and 4 C

```
(defrule low-temp
       (temp ?day ?temp&
```

Example : summing up the area of a group of rectangles

- The facts defined in deffacts are asserted into the knowledge base whenever a reset command is issued:

```
(deftemplate rectangle (slot height) (slot width))
(deffacts initial-information
  (rectangle (height 10) (width 6))
  (rectangle (height 5) (width 3))
  (rectangle (height 2) (width 5))
  (sum 0))
```

Sum-rectangles

```
(defrule sum-rectangles
  (rectangle (height ?h) (width ?w))
  ?sum <- (sum ?total)
  =>
  (retract ?sum)
  (assert (sum (+ ?total (* ?h ?w)))))
```

Ok?

Loops endlessly because of new sum fact…
Solution?
Retract rectangle fact …   or
Use temporary fact that can be retracted

Sum-rectangles: corrected

```
(defrule sum-rectangles
  (rectangle (height ?h) (width ?w))
  =>
  (assert (add-to-sum (* ?h ?w))))

(defrule sum-areas
  ?sum <- (sum ?total)
  ?newa <- (add-to-sum ?na)
  =>
  (retract ?newa ?sum)
  (assert (sum (+ ?total ?na))))
Jess> (watch all)
```

Jess: donor example

```
(clear)
(deftemplate blood-donor (slot name) (slot type))
(deffacts blood-bank
       ; put names & their types into working memory
  (blood-donor (name "Alice")(type "A"))
  (blood-donor (name "Agatha")(type "A"))
  (blood-donor (name "Bob")(type "B"))
  (blood-donor (name "Barbara")(type "B"))
  (blood-donor (name "Jess")(type "AB"))
  (blood-donor (name "Karen")(type "AB"))
  (blood-donor (name "Onan")(type "O"))
  (blood-donor (name "Osbert")(type "O")) )
(defrule can-give-to-same-type-but-not-self
  ; handles A > A, B > B, O > O, AB > AB, but not N1 > N1
  (blood-donor (name ?name)(type ?type))
  (blood-donor (name ?name2)(type ?type2 &:(eq ?type
                  ?type2) &: (neq ?name ?name2) ))
 => (printout t ?name " can give blood to " ?name2 crlf) )
```

Jess: donor example

```
(defrule O-gives-to-others-but-not-itself
 ; O to O covered in above rule
  (blood-donor (name ?name)(type ?type &:(eq ?type "O")))
  (blood-donor (name ?name2)(type ?type2 &: (neq ?type
   ?type2) &: (neq ?name ?name2) ))
  => (printout t ?name " can give blood to " ?name2 crlf) )

(defrule A-or-B-gives-to-AB
 ; case O gives to AB and AB gives to AB already dealt with
  (blood-donor (name ?name)(type ?type &:(or (eq ?type "A")
   (eq ?type "B" ))))
  (blood-donor (name ?name2)(type ?type2 &: (eq ?type2 "AB")
   &: (neq ?name ?name2) ))
  => (printout t ?name " can give blood to " ?name2 crlf) )
;(watch all)
(reset)
(run)
```

What is going on??

```
Jess> (batch “blood.clp”)
TRUE
Jess> (facts)
For a total of 0 facts in module MAIN.
Jess> (reset)
 ==> Focus MAIN
 ==> f-0 (MAIN::initial-fact)
 ==> f-1 (MAIN::blood-donor (name "Alice") (type "A"))
 ==> f-2 (MAIN::blood-donor (name "Agatha") (type "A"))
==> Activation: MAIN::can-give-to-same-type-but-not-self :  f-2, f-1
==> Activation: MAIN::can-give-to-same-type-but-not-self :  f-1, f-2
 ==> f-3 (MAIN::blood-donor (name "Bob") (type "B"))
 ==> f-4 (MAIN::blood-donor (name "Barbara") (type "B"))
==> Activation: MAIN::can-give-to-same-type-but-not-self :  f-4, f-3
==> Activation: MAIN::can-give-to-same-type-but-not-self :  f-3, f-4
 ==> f-5 (MAIN::blood-donor (name "Jess") (type "AB"))
==> Activation: MAIN::A-or-B-gives-to-AB :  f-1, f-5
==> Activation: MAIN::A-or-B-gives-to-AB :  f-2, f-5
==> Activation: MAIN::A-or-B-gives-to-AB :  f-3, f-5
==> Activation: MAIN::A-or-B-gives-to-AB :  f-4, f-5
```

What is going on?? (2)

```
==> f-6 (MAIN::blood-donor (name "Karen") (type "AB"))
==> Activation: MAIN::can-give-to-same-type-but-not-self :  f-6, f-5
==> Activation: MAIN::can-give-to-same-type-but-not-self :  f-5, f-6
==> Activation: MAIN::A-or-B-gives-to-AB :  f-1, f-6
==> Activation: MAIN::A-or-B-gives-to-AB :  f-2, f-6
==> Activation: MAIN::A-or-B-gives-to-AB :  f-3, f-6
==> Activation: MAIN::A-or-B-gives-to-AB :  f-4, f-6
  ==> f-7 (MAIN::blood-donor (name "Onan") (type "O"))
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-7, f-1
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-7, f-2
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-7, f-3
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-7, f-4
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-7, f-5
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-7, f-6
 ==> f-8 (MAIN::blood-donor (name "Osbert") (type "O"))
==> Activation: MAIN::can-give-to-same-type-but-not-self :  f-8, f-7
==> Activation: MAIN::can-give-to-same-type-but-not-self :  f-7, f-8
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-8, f-1
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-8, f-2
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-8, f-3
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-8, f-4
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-8, f-5
==> Activation: MAIN::O-gives-to-others-but-not-itself :  f-8, f-6
TRUE
```

What is going on?? (3)

```
Jess> (run)
FIRE 1 MAIN::can-give-to-same-type-but-not-self f-7, f-8
Onan can give blood to Osbert
FIRE 2 MAIN::can-give-to-same-type-but-not-self f-8, f-7
Osbert can give blood to Onan
FIRE 3 MAIN::O-gives-to-others-but-not-itself f-8, f-6
Osbert can give blood to Karen
FIRE 4 MAIN::O-gives-to-others-but-not-itself f-8, f-5
Osbert can give blood to Jess
FIRE 5 MAIN::O-gives-to-others-but-not-itself f-8, f-4
Osbert can give blood to Barbara
FIRE 6 MAIN::O-gives-to-others-but-not-itself f-8, f-3
Osbert can give blood to Bob
FIRE 7 MAIN::O-gives-to-others-but-not-itself f-8, f-2
Osbert can give blood to Agatha
FIRE 8 MAIN::O-gives-to-others-but-not-itself f-8, f-1
Osbert can give blood to Alice
FIRE 9 MAIN::O-gives-to-others-but-not-itself f-7, f-6
Onan can give blood to Karen
FIRE 10 MAIN::O-gives-to-others-but-not-itself f-7, f-5
Onan can give blood to Jess
```

What is going on?? (4)

```
FIRE 11 MAIN::O-gives-to-others-but-not-itself f-7, f-4
Onan can give blood to Barbara
FIRE 12 MAIN::O-gives-to-others-but-not-itself f-7, f-3
Onan can give blood to Bob
FIRE 13 MAIN::O-gives-to-others-but-not-itself f-7, f-2
Onan can give blood to Agatha
FIRE 14 MAIN::O-gives-to-others-but-not-itself f-7, f-1
Onan can give blood to Alice
FIRE 15 MAIN::can-give-to-same-type-but-not-self f-5, f-6
Jess can give blood to Karen
FIRE 16 MAIN::can-give-to-same-type-but-not-self f-6, f-5
Karen can give blood to Jess
FIRE 17 MAIN::A-or-B-gives-to-AB f-4, f-6
Barbara can give blood to Karen
FIRE 18 MAIN::A-or-B-gives-to-AB f-3, f-6
Bob can give blood to Karen
FIRE 19 MAIN::A-or-B-gives-to-AB f-2, f-6
Agatha can give blood to Karen
FIRE 20 MAIN::A-or-B-gives-to-AB f-1, f-6
Alice can give blood to Karen
```

What is going on?? (5)

```
FIRE 21 MAIN::A-or-B-gives-to-AB f-4, f-5
Barbara can give blood to Jess
FIRE 22 MAIN::A-or-B-gives-to-AB f-3, f-5
Bob can give blood to Jess
FIRE 23 MAIN::A-or-B-gives-to-AB f-2, f-5
Agatha can give blood to Jess
FIRE 24 MAIN::A-or-B-gives-to-AB f-1, f-5
Alice can give blood to Jess
FIRE 25 MAIN::can-give-to-same-type-but-not-self f-3, f-4
Bob can give blood to Barbara
FIRE 26 MAIN::can-give-to-same-type-but-not-self f-4, f-3
Barbara can give blood to Bob
FIRE 27 MAIN::can-give-to-same-type-but-not-self f-2, f-1
Agatha can give blood to Alice
FIRE 28 MAIN::can-give-to-same-type-but-not-self f-1, f-2
Alice can give blood to Agatha
 <== Focus MAIN
28
```

Order of firing rules

```
(defrule fire-first
  (priority first)
  =>
  (printout t “first rule” crlf))
(defrule fire-second
  (priority second)
  =>
  (printout t “second rule” crlf))

(defrule fire-third
  (priority third)
  =>
  (printout t “third rule” crlf))
```

Order …

```
Jess> (reset)
(assert (priority first))
(assert (priority second))
(assert (priority third))

(run)
third rule
second rule
first rule

% activated rules are put on a stack; then first
  rule on top of the stack …
```

- Data
- Rules
- Inference engine

The agenda

- Set of activated rules = conflict set
- Conflict-resolution strategies:
  - Depth (default): most recently activated rule fires first
  - Breadth (set-strategy breadth)
  - Writing your own strategy is possible
- More fine-tuning by using salience

Order and salience (discouraged!!)

```
(defrule fire-first
   (declare salience 30)    ; before the salience 20 ones
   (priority first)
   =>
   (printout t “first rule” crlf))
(defrule fire-second
   (declare salience 20)
   (priority second)
   =>
   (printout t “second rule” crlf))

(defrule fire-third
   (declare salience 20)
   (priority third)
   =>
   (printout t “third rule” crlf))
```

Levels in RB programs

- Control rules ; organize phases
- Query rules ; ask extra info to user
- Expert rules ;
- Constraint rules ; detect illegal states

What with their salience???

(from low (control rules) to high (constraint rules)

- Also modules

Rule evaluation: naive way

- A fixed set of rules (R) while the knowledge base changes continuously.
- Number of facts (F) +- with P is the average number of patterns per rule LHS
- Naive: keep a list of rules and continuously cycle through the list, checking each one's left-hand-side (LHS) against the knowledge base and executing the right-hand-side (RHS) of any rules that apply.
- You might call this the *rules finding facts* approach and its computational complexity is of the order of O(R(F^P))
- Note: CHR is facts finding rules (active constraint).

Rule evaluation: Rete

- *Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19 (1982), 17-37)*
- Basis for a whole generation of fast expert system shells: OPS5, its descendant ART, and CLIPS.
- Remembering past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs.
- As a result, the computational complexity per iteration drops to something more like O(RFP), or linear in the size of the fact base.

Rule evaluation: Rete

- Rete is implemented by building a network of nodes, each of which represents one or more tests found on a rule LHS.
- Facts that are being added to or removed are processed by this network of nodes.
- At the bottom of the network are nodes representing individual rules.
- When a set of facts filters all the way down to the bottom of the network, it has passed all the tests on the LHS of a particular rule and this set becomes an *activation*.
- The associated rule may have its RHS executed (*fired*) if the activation is not invalidated first by the removal of one or more facts from its activation set.

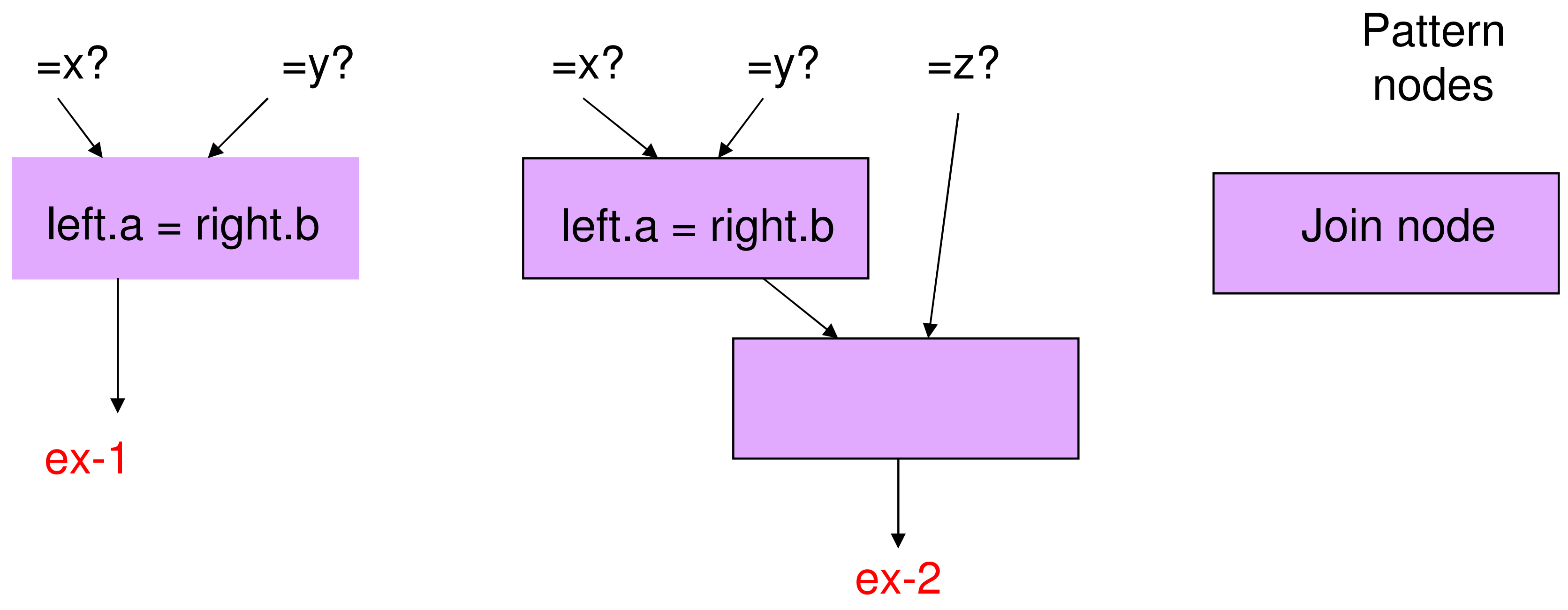
Example

```
(deftemplate x (slot a))
(deftemplate y (slot b))
(deftemplate z (slot c))
(defrule ex-1 (x (a ?v1)) (y (b ?v1)) => )
(defrule ex-2 (x (a ?v2)) (y (b ?v2)) (z)  => )

Pattern nodes: =x?          =y?
Join node on pattern nodes: left.a=right.b?
Terminal node: represent the individual rules (ex-1)
```

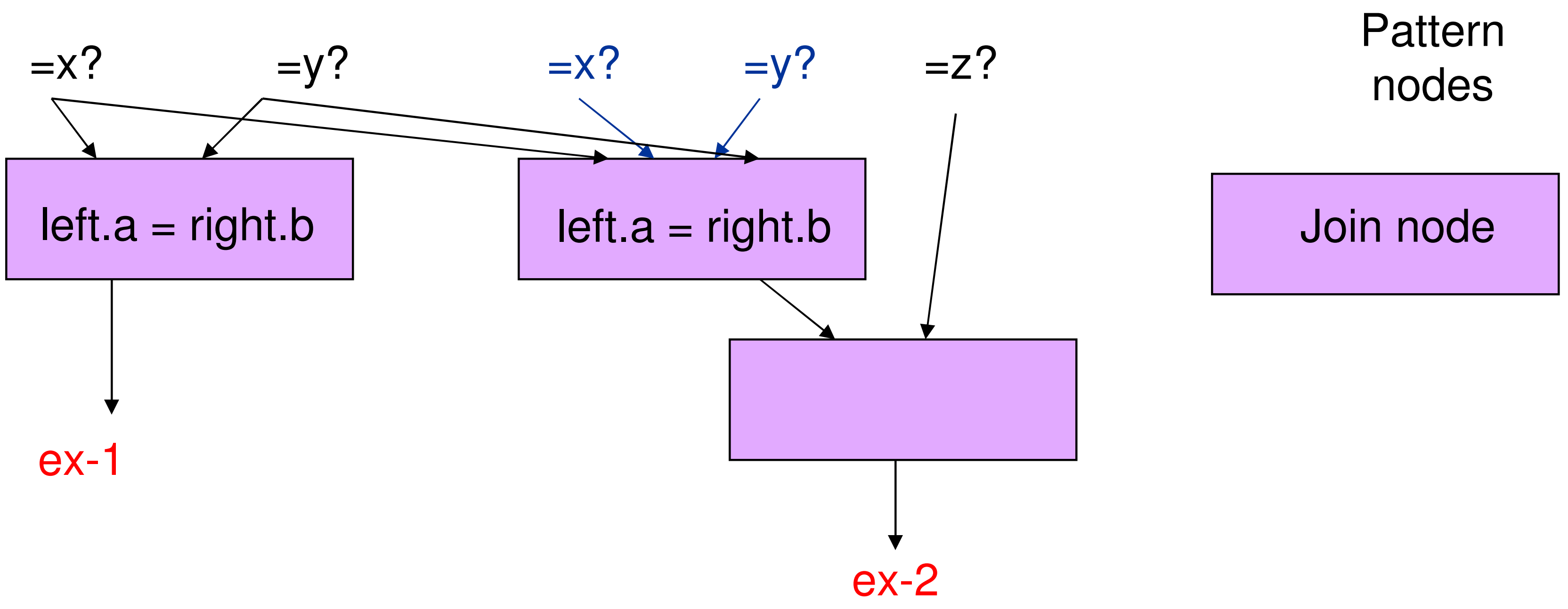
Example

```
(defrule ex-1 (x (a ?v1)) (y (b ?v1)) => )
(defrule ex-2 (x (a ?v2)) (y (b ?v2)) (z)  => )
```



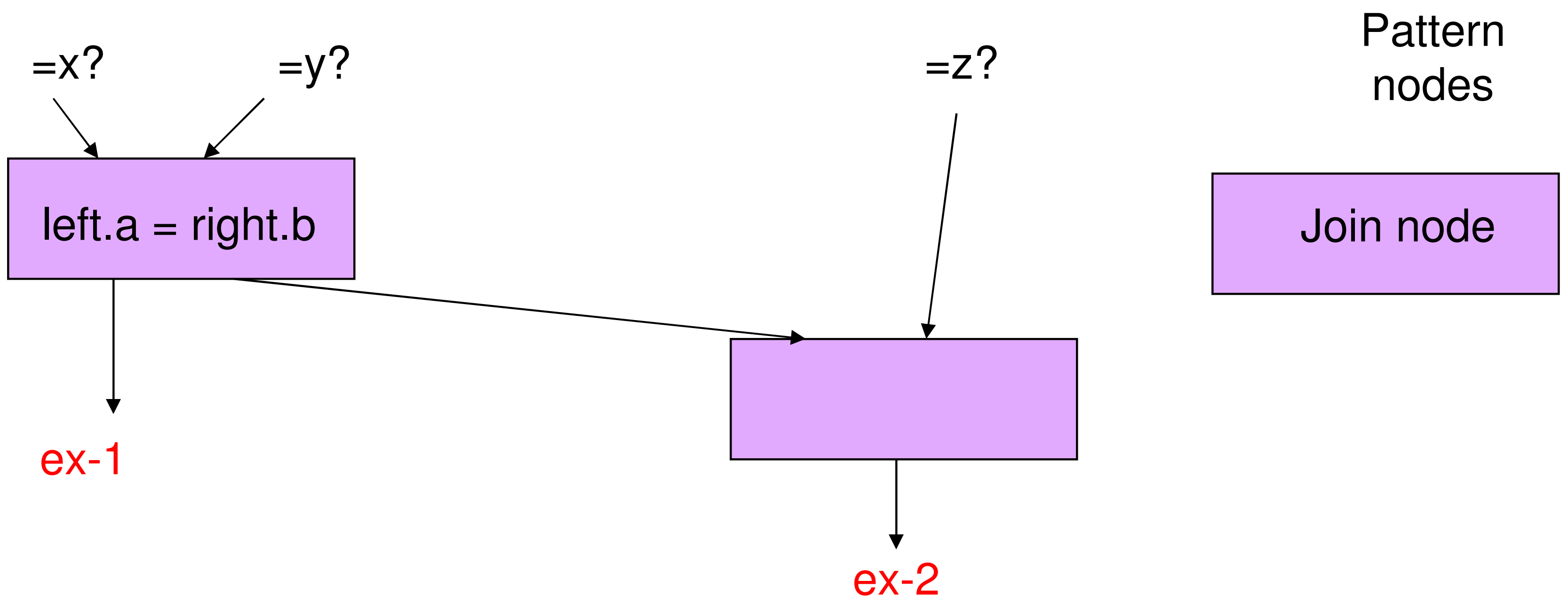
Example: optimizations

```
(defrule ex-1 (x (a ?v1)) (y (b ?v1)) => )
(defrule ex-1 (x (a ?v2)) (y (b ?v2)) (z)  => )
```



Example: optimizations

```
(defrule ex-1 (x (a ?v1)) (y (b ?v1)) => )
(defrule ex-1 (x (a ?v2)) (y (b ?v2)) (z)  => )
```



Finding matching facts

- When facts are added (or removed), which patterns have been matched?
  - Pattern network, one-input nodes
  - Alpha (right) memories: all facts that matched a pattern
- Comparison of variable bindings across patterns: partial matches
  - Join network, two-input nodes
  - Beta (left) memories
  - rule with N patterns: N-1 joins

Rete

- Uses memory for speed!!
- (What is done by CHR???)
- Memories retain information about success or failure of pattern matches during the previous cycles.
- Takes advantage of the temporal redundancy (the rules change only a few facts) and the structural similarity between rules.
- Special nodes for not and test

- Importance of pattern order!!
- What is stored by Rete???
  What should be avoided??

Simple program: partial matches??

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

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

Simple program: Rete network??

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

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

Ordering patterns for efficiency

- Most specific patterns go first
- Pattern matching volatile facts go last
- Patterns matching the fewest facts go first

- Put the tests as soon as possible

Agenda example

```
(deffacts initf
     (item x a)
     (item x b)
     (item x c)
)

(defrule comb-1
    (item ?x ?y)
    (item ?x ?z&~?y)
    =>
    (assert (b ?y ?z))

(defrule print-b
    (b ?y ?z)
    =>
    (printout t "b found " ?y " "  ?z  crlf)
)
```

```
Jess> (reset)
 ==> Focus MAIN
 ==> f-0 (MAIN::initial-fact)
 ==> f-1 (MAIN::item x a)
 ==> f-2 (MAIN::item x b)
==> Activation: MAIN::comb-1 :  f-2, f-1
==> Activation: MAIN::comb-1 :  f-1, f-2
 ==> f-3 (MAIN::item x c)
==> Activation: MAIN::comb-1 :  f-3, f-1
==> Activation: MAIN::comb-1 :  f-3, f-2
==> Activation: MAIN::comb-1 :  f-1, f-3
==> Activation: MAIN::comb-1 :  f-2, f-3
TRUE
```

Agenda example

```
(deffacts initf
     (item x a)
     (item x b)
     (item x c)
)

(defrule comb-1
    (item ?x ?y)
    (item ?x ?z&~?y)
    =>
    (assert (b ?y ?z))

(defrule print-b
    (b ?y ?z)
    =>
    (printout t "b found " ?y " "  ?z  crlf)
)
```

```
Jess> (run)
FIRE 1 MAIN::comb-1 f-3, f-2
 ==> f-4 (MAIN::b c b)
==> Activation: MAIN::print-b :  f-4
FIRE 2 MAIN::print-b f-4
b found c b
FIRE 3 MAIN::comb-1 f-2, f-3
 ==> f-5 (MAIN::b b c)
==> Activation: MAIN::print-b :  f-5
FIRE 4 MAIN::print-b f-5
b found b c
FIRE 5 MAIN::comb-1 f-3, f-1
 ==> f-6 (MAIN::b c a)
==> Activation: MAIN::print-b :  f-6
FIRE 6 MAIN::print-b f-6
b found c a
```

Agenda example

```
(deffacts initf
     (item x a)
     (item x b)
     (item x c)
)

(defrule comb-1
    (item ?x ?y)
    (item ?x ?z&~?y)
    =>
    (assert (b ?y ?z))

(defrule print-b
    (b ?y ?z)
    =>
    (printout t "b found " ?y " "  ?z  crlf)
)
```

```
…
FIRE 9 MAIN::comb-1 f-2, f-1
 ==> f-8 (MAIN::b b a)
==> Activation: MAIN::print-b :  f-8
FIRE 10 MAIN::print-b f-8
b found b a
FIRE 11 MAIN::comb-1 f-1, f-2
 ==> f-9 (MAIN::b a b)
==> Activation: MAIN::print-b :  f-9
FIRE 12 MAIN::print-b f-9
b found a b
 <== Focus MAIN
12
Jess>
```

Agenda extra rule

```
(defrule comb-2
    (b ?y1 ?z1&~?y1)
    (b ?z1 ?y1)
    =>
    (assert (comb-2 ?y1 ?z1))
)
(defrule comb-1
    (item ?x ?y)
    (item ?x ?z&~?y)
    =>
    (assert (b ?y ?z))

(defrule print-b
    (b ?y ?z)
    =>
    (printout t "b found " ?y " "  ?z  crlf)
)
```

```
Jess>(reset)
..
Jess> (run)  ;;;; when rule com-2??
FIRE 1 MAIN::comb-1 f-3, f-2
 ==> f-4 (MAIN::b c b)
==> Activation: MAIN::print-b :  f-4
FIRE 2 MAIN::print-b f-4
b found c b
FIRE 3 MAIN::comb-1 f-2, f-3
 ==> f-5 (MAIN::b b c)
==> Activation: MAIN::print-b :  f-5 ;;
FIRE 4 MAIN::print-b f-5
b found b c
FIRE 5 MAIN::comb-1 f-3, f-1
```

Agenda extra rule

```
(defrule comb-2
    (b ?y1 ?z1&~?y1)
    (b ?z1 ?y1)
    =>
    (assert (comb-2 ?y1 ?z1))
)
(defrule comb-1
    (item ?x ?y)
    (item ?x ?z&~?y)
    =>
    (assert (b ?y ?z))

(defrule print-b
   …
```

```
Jess> (run)  ;;;; when rule com-2??
FIRE 1 MAIN::comb-1 f-3, f-2
 ==> f-4 (MAIN::b c b)
==> Activation: MAIN::print-b :  f-4
FIRE 2 MAIN::print-b f-4
b found c b
FIRE 3 MAIN::comb-1 f-2, f-3
 ==> f-5 (MAIN::b b c)
==> Activation: MAIN::comb-2 :  f-5, f-4
==> Activation: MAIN::comb-2 :  f-4, f-5
==> Activation: MAIN::print-b :  f-5
FIRE 4 MAIN::comb-2 f-5, f-4
 ==> f-6 (MAIN::comb-2 b c)
FIRE 5 MAIN::comb-2 f-4, f-5
 ==> f-7 (MAIN::comb-2 c b)
FIRE 6 MAIN::print-b f-5
b found b c
FIRE 7 MAIN::comb-1 f-3, f-1
```

Exercise: find the largest number

```
(deffacts max-num
  (loop-max 100))
(defrule loop-assert
  (loop-max ?n)
  =>
  (bind ?i 1)
  (while (<= ?i ?n)
     (assert (number ?i))
     (bind ?i (+ ?i 1))))
```

```
(defrule largest-number
  (number ?n1)
  (not  (number ?n2&:(> ?n2 ?n1)))
  =>
  (printout t ?n1 crlf))
```

Logical CE

-
  ```
  (defrule ok
       (logical (a))
       (logical (b))
       (c)
        => (assert (d)))
  ```
- (d) is made logically dependent upon the LHS of the rule.
  (a) and (b) provide **logical support** to (d)
  If (a) or (b) is removed, then also (d) is removed.

forall Conditional Expression!!!

- The "forall" grouping CE matches if, for every match of the first pattern inside it, all the subsequent patterns match.

```
(defrule every-employee-has-a-stapler-and-holepunch
  (forall (employee (name ?n))
          (stapler (owner ?n)) (holepunch (owner ?n)))
  => (printout t "Every employee has a stapler and a
  holepunch." crlf))
```

- This rule fires if there are a hundred employees and everyone owns the appropriate supplies. If a single employee doesn't own the supplies, the rule won't fire.

Backward chaining

- Have to declare which facts can serve as backward-chaining triggers
- To pull required data into Jess's working memory from a database on demand (or from the user)
- To avoid redundant computation

```
Jess> (do-backward-chaining factorial)   ;;;!!!!
Jess> (defrule print-factorial-10)
        (factorial 10 ?r1)
         =>
         (printout t “The factorial of 10 is “ ?r1 crlf)
```

Backward chaining

- Due to backward chaining Jess asserts

```
(need-factorial 10 nil)
```

```
Jess> (defrule do-factorial
           (need-factorial ?x ?)  ;; trigger fact
           =>
           (bind ?r 1)
           (bind ?n ?x)
            (while (> ?n 1)
               (bind ?r (* ?r ?n))
               (bind ?n (- ?n 1)))
            (assert (factorial ?x ?r)))
```

Jess Programming

- Actually a production-rule system
- To encode human reasoning rules
- To solve problems
- To implement search problems using a state space represention
  - Working memory has the states that have been reached
  - Needs to store information in order to avoid loops

For more details: see wiki

- Website of Jess
- Book Jess in Action
  - Ernest Friedman-Hill
- Mailing list of Jess users

- **Jess Meets Einstein’s Riddle**

http://www.developer.com/java/other/article.php/3089641

Riddle: 4 golfers on a row; clues

- a link between a name and either a position or a color:

```
(deftemplate pants-color (slot of) (slot is))
(deftemplate position (slot of) (slot is))
```

- A pants-color fact represents the idea that one specific golfer (named in the of slot) has a certain color pants (named in the is slot.)

Riddle: golfers on a row;

- You‘ll use these templates to create facts representing each of the possible combinations. There are 32 of them altogether—for example:

```
(pants-color (of Bob) (is red)) ; 4 * 4
(position (of Joe) (is 3))      ; 4 * 4
```

Riddle snapshots

- You can write a rule to create all 32 of these facts and put them into *working memory:*

```
(defrule generate-possibilities
  =>
  (foreach ?name (create$ Fred Joe Bob Tom)
     (foreach ?color (create$ red blue plaid orange)
        (assert (pants-color (of ?name)
                 (is ?color))))
     (foreach ?position (create$ 1 2 3 4)
        (assert (position (of ?name) (is ?position))))))
```

Riddle snapshots

- Here's the first useful sentence, *The golfer to Fred's immediate right is wearing blue pants*:

```
(defrule find-solution
;; There is a golfer named Fred, whose position is ?p1
;; and pants color is ?c1
  (position (of Fred) (is ?p1))
  (pants-color (of Fred) (is ?c1))
;; The golfer to Fred's immediate right
;; is wearing blue pants.
  (position (of ?n&~Fred) (is ?p&:(eq ?p (+ ?p1 1))))
  (pants-color (of ?n&~Fred) (is blue&~?c1))
…    => !!!
```

The end