Jess Backward Chaining

The Jess Language Part 5
CS3025, Knowledge-Based Systems
Lecture 16

Yuting Zhao
Yuting.zhao@gmail.com

2017-11-09

Outline

- Variable Binding in Condition Elements
- Refraction
- Jess – Backward Chaining

Variable Binding in Condition Elements

Problem with OR

Condition Element Operators

- Remember:
  - Patterns appear on the LHS of a rule
  - Operators "**and**", "or", "not" are operators over complete patterns

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

Is the same !!!

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

This LHS specifies:
if both patterns match, then the rule is activated

The Problem with "or"

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

Wrong !!!

- Operator "or":
  - The rule fires, if
    - Either: The first pattern matches
    - Or: The second pattern matches
    - Or: Both pattern match
- Problem !!
  - ?x will have binding only if first pattern matches !!!
  - Therefore: a rule cannot be defined like that !!!

The Problem with “or”

- Careful in case of “or”:
  - If we use variable ?x at the RHS, it has to occur in all the patterns !!!

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

Correct

Refraction

Avoiding Infinite Loops via unwanted
re-activation of rules

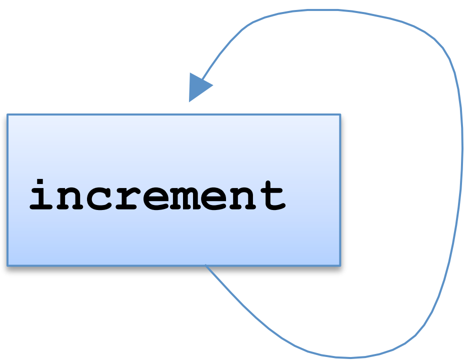
Observation

- Investigate the behaviour of the following rule:

```
(deftemplate person
     (slot name)
     (slot salary))

(defrule increment
     ?f <- (person
               (name   ?name)
               (salary ?salary))
     =>
     (modify ?f (salary (+ ?salary 10)))
)
```

In Jess, loops are not only caused by “while”, but also by a simple rule



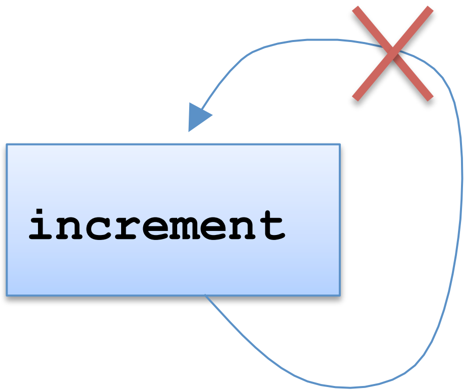
fact: (person (name Tom) (salary 100))

Refraction
Avoid Re-activation of a Rule

- **Refraction** is the act of preventing a rule from re-activating itself

- Declaration in rule:

```
(defrule increment
    (declare (no-loop TRUE))
    ?f <- (person
              (name    ?name)
              (salary ?salary))
    =>
    (modify ?f (salary (+ ?salary 10)))
)
```



Another example

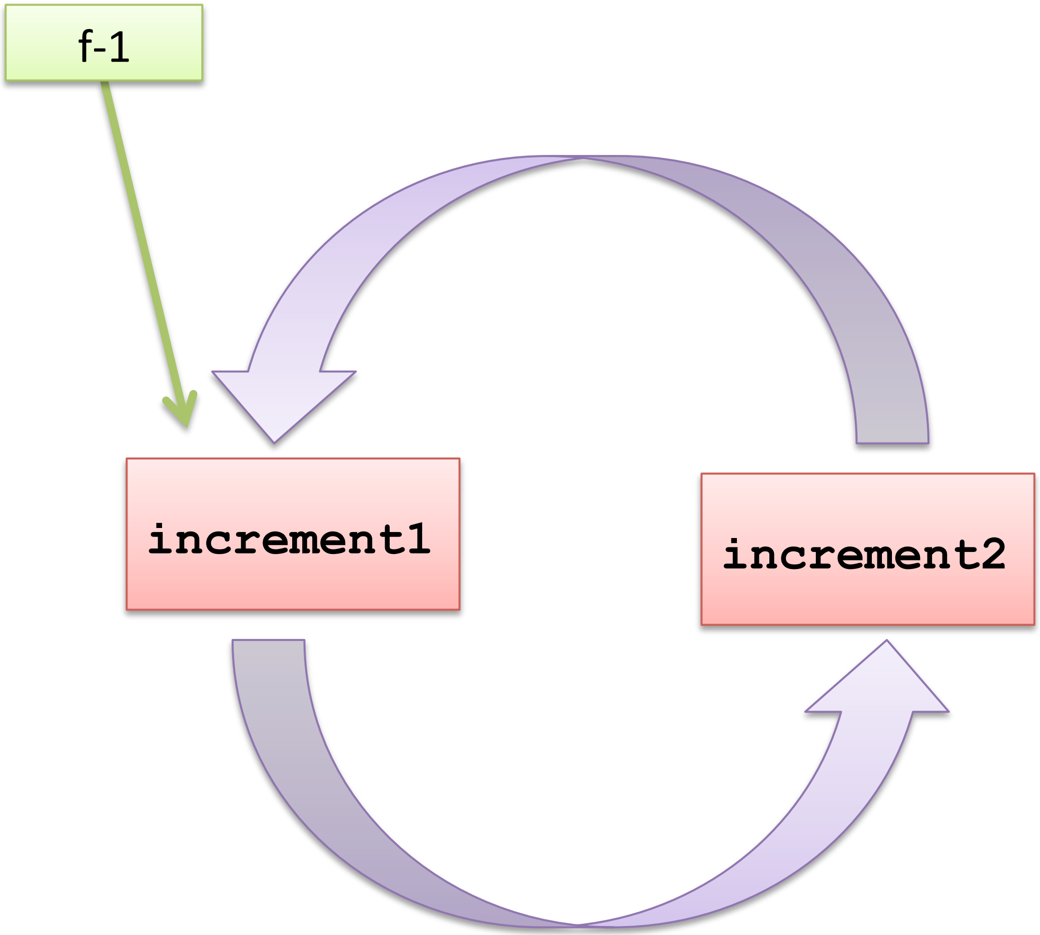
```
(deftemplate person     (slot name)    (slot age)     (slot salary) )

(deffacts persons
   (person (name Tom) (age 10) (salary 100))
)

(defrule increment1
   (declare (no-loop TRUE))
   ?f <- (person   (name   ?name)  (age 10)  (salary ?salary))
   =>
   (modify ?f (salary (+ ?salary 10)))
   (modify ?f (age 9))
   (printout t ?name " age 10 has salary "  ?salary crlf)
)

(defrule increment2
   (declare (no-loop TRUE))
   ?f <- (person    (name   ?name)  (age 9)  (salary ?salary))
   =>
   (modify ?f (salary (+ ?salary 10)))
   (modify ?f (age 10))
   (printout t ?name " age 9 has salary "  ?salary crlf)
)
```

Ping-pang loop



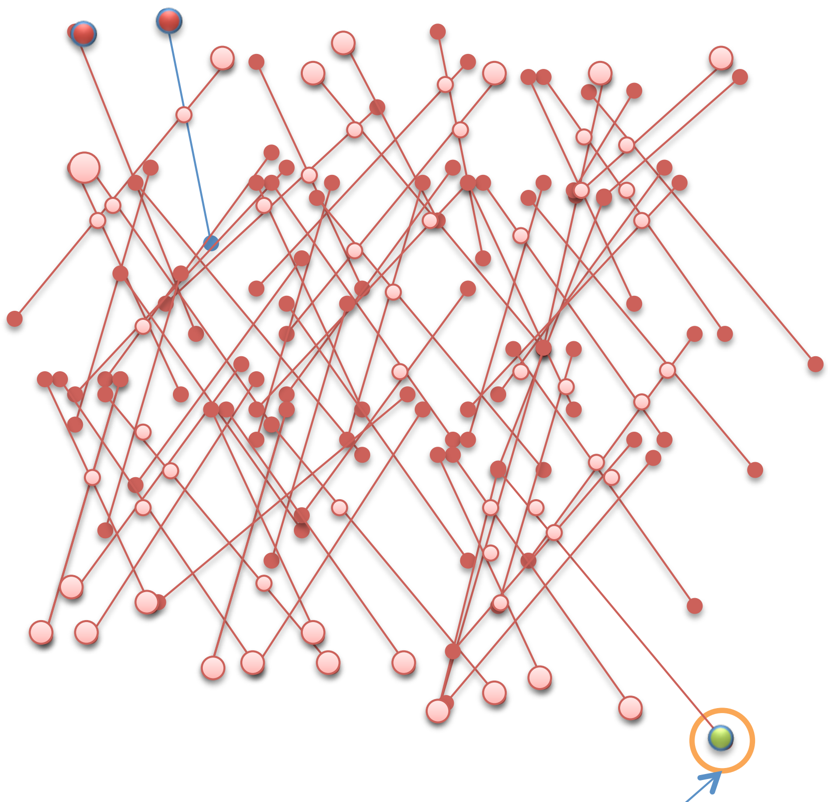

```
Jess> (reset)
Jess> (run)
Tom age 10 has salary 100
Tom age 9 has salary 110
Tom age 10 has salary 120
Tom age 9 has salary 130
Tom age 10 has salary 140
Tom age 9 has salary 150
Tom age 10 has salary 160
Tom age 9 has salary 170
Tom age 10 has salary 180
Tom age 9 has salary 190
Tom age 10 has salary 200
Tom age 9 has salary 210
Tom age 10 has salary 220
……..
Terminate batch job (Y/N)? y
```

Ctrl + c

Challenge: could you design a **triangle loop**?

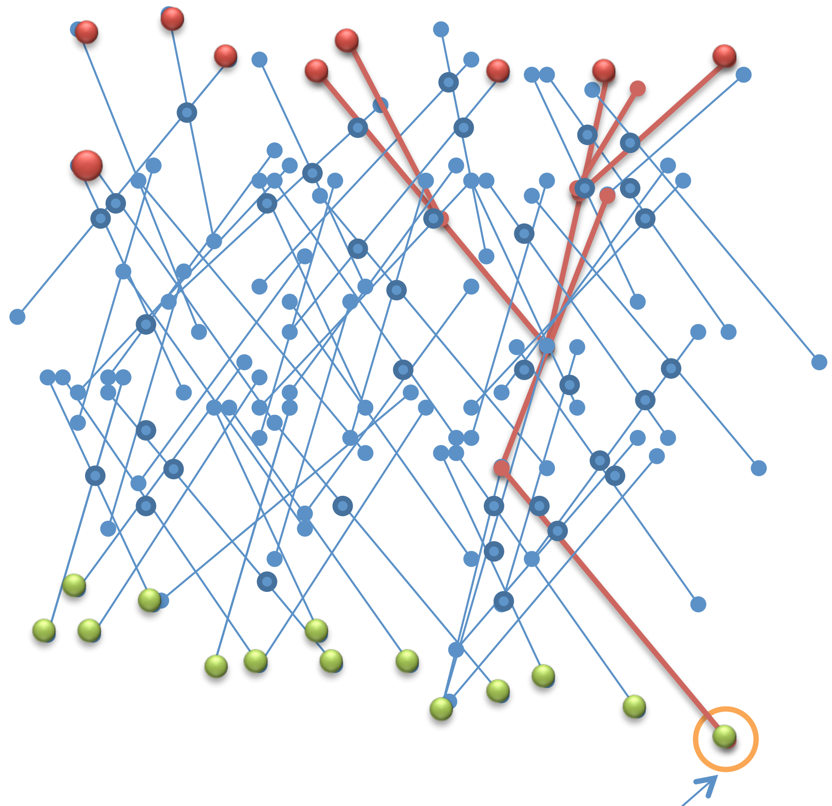
Jess – Backward Chaining

Backward Chaining (motivation)



Could we prove this?

Forward chaining checks all facts and rules to prove it



Could we prove this?

Backward chaining checks only related facts and rules to prove it

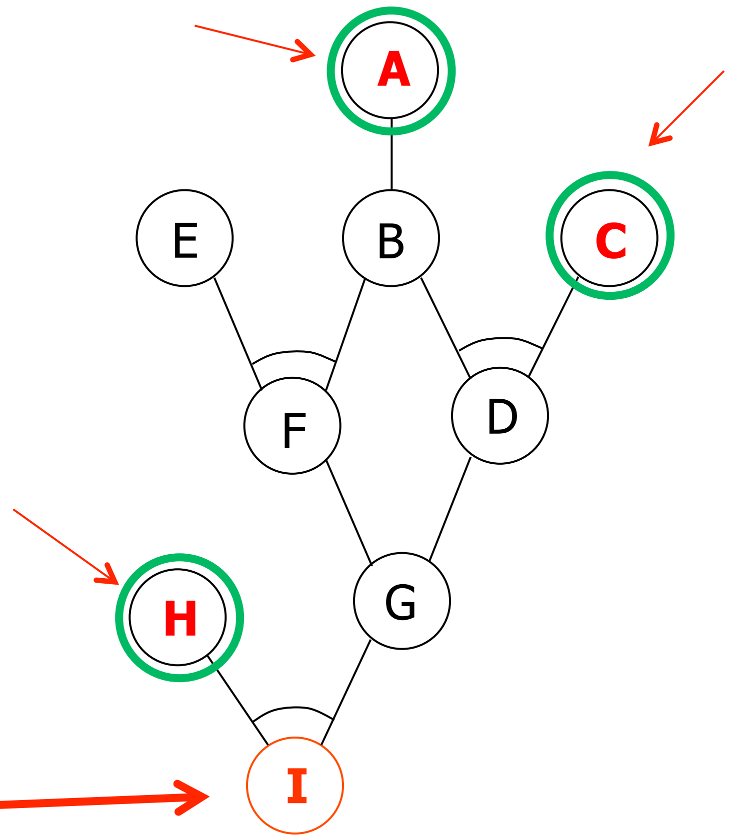
Backward Chaining

- Given a particular **hypothesis** (a fact we regard as true), which we also call a "goal"
  - We want to prove that the hypothesis is true
- For each rule whose **consequent** matches the current hypothesis:
  - Find support (matching facts) for the rule's **antecedents**
    - Match them to known facts (in WM)
    - Chain backwards through other rules that create hypotheses that, if proven, will support antecedents of your current rule
  - If all the of the current rule's antecedents are supported (match facts or supported hypotheses), then we can conclude that its consequent (and our original hypothesis) is true
- Classical language based on backward chaining: Prolog

- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

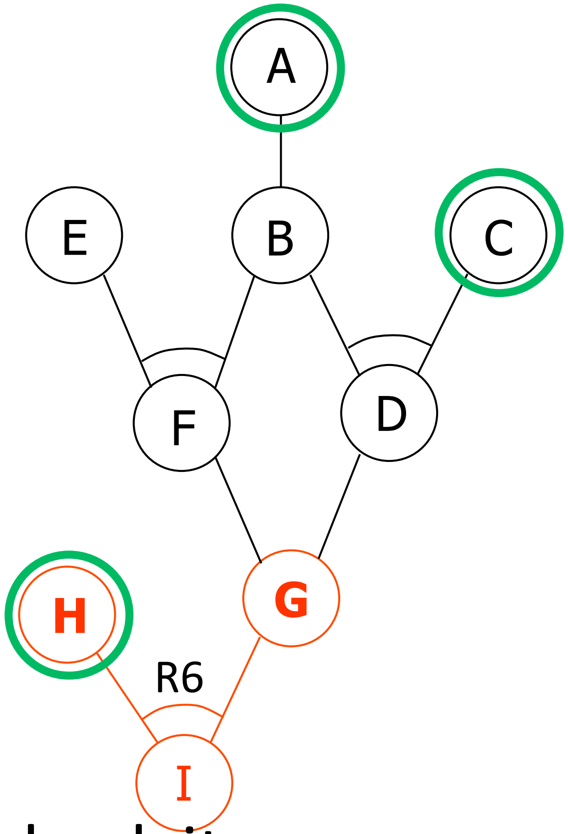
In WM: A,C and H

Our Goal: I



- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

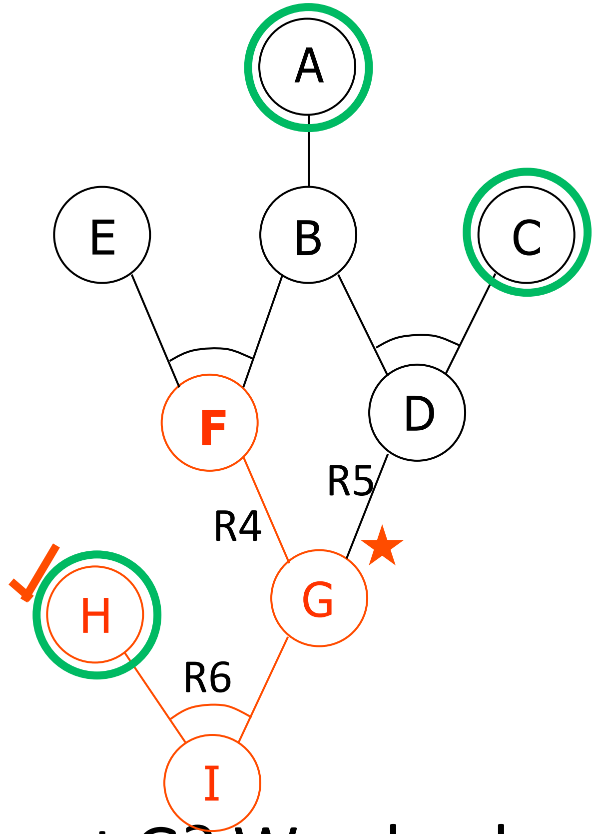
__In WM__: A,C and H



- I is not known to be true, then check it
- Rule R6 supports I, but only if H and G are true
- We regard H and G as new "**sub-goals**" to be achieved

- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

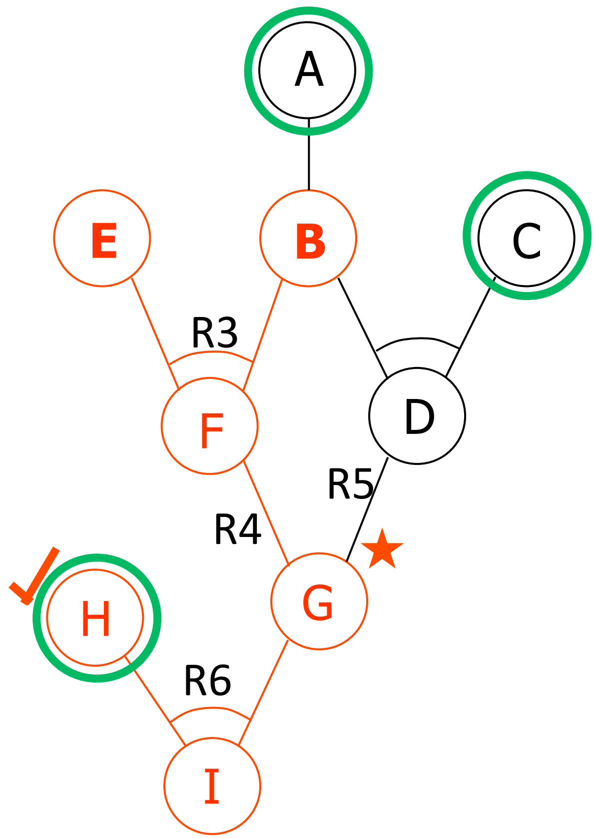
<u>*In WM*</u>: A,C and H



- H is known to be true, what about G? We check.
- Rules R4 and R5 support G – choose R4 and mark R5 for ***backtracking*** (if backtracking – then back to here)

- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

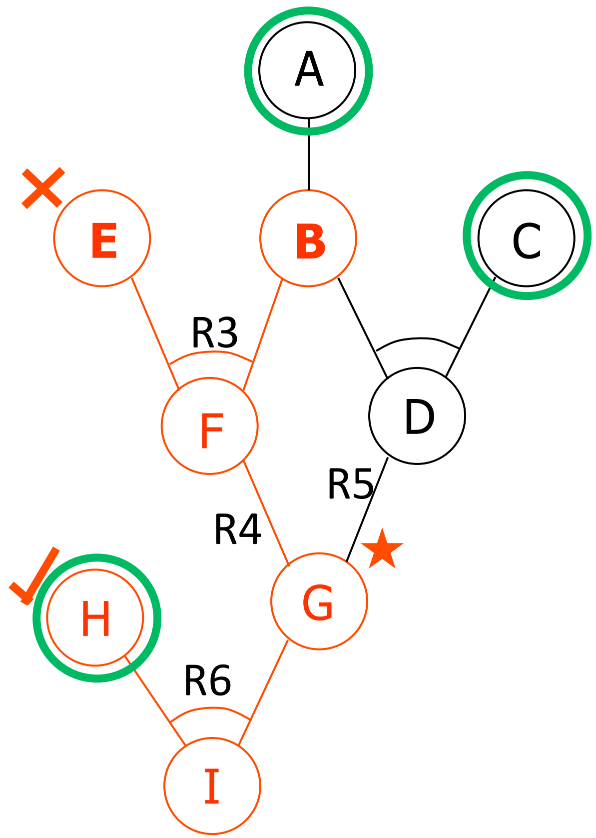
<u>In WM</u>: A,C and H



- F is not known to be true. We check if it is supported.
- Rule R3 supports F – post E and B as goals

- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

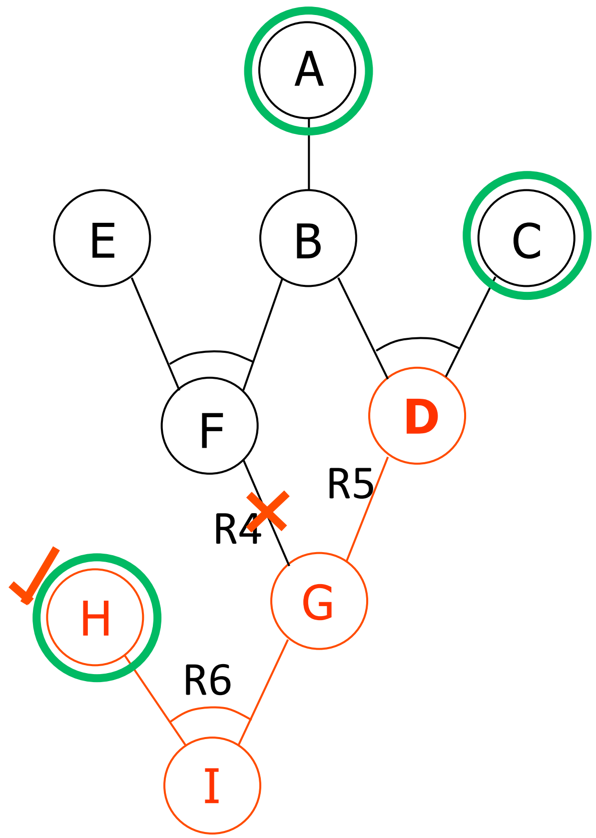
<u>In WM</u>: A,C and H



- E is not known to be true. We check if it is supported.
- **No rule supports E**
- ***Backtrack !!***

- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

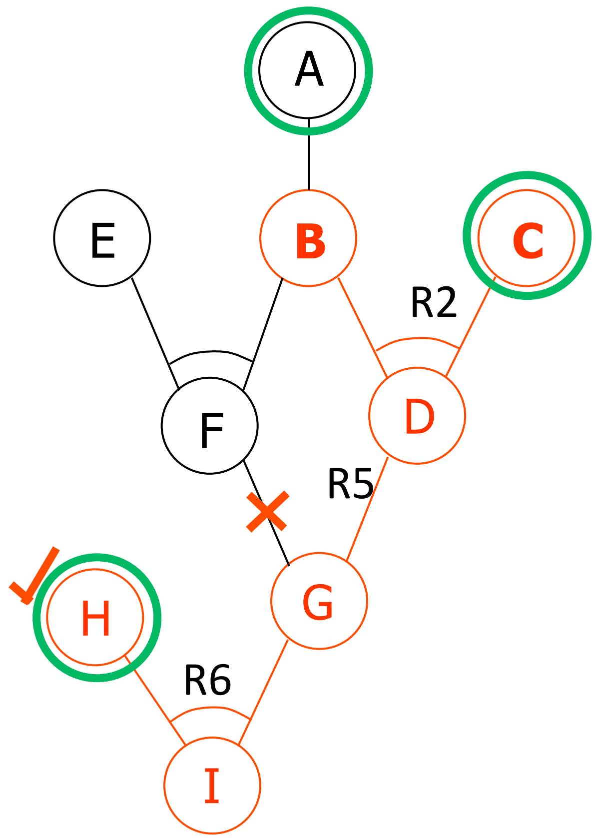
<u>In WM</u>: A,C and H



- Backtrack to G. It is unknown. We check if it is supported.
- Rule R5 supports G – post D as a goal

- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

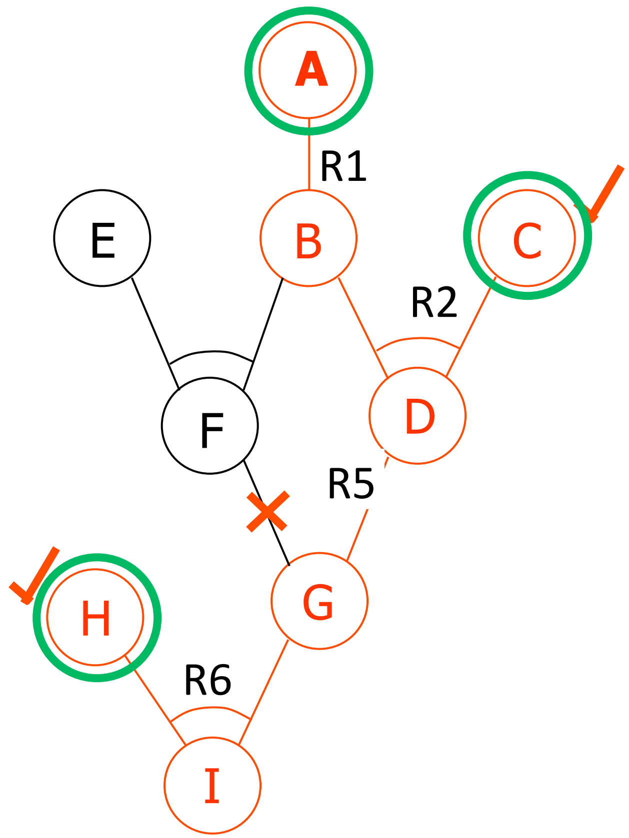
In WM: A,C and H



- D is not known to be true. We check it.
- Rule R2 supports D – post B and C as a goals

- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

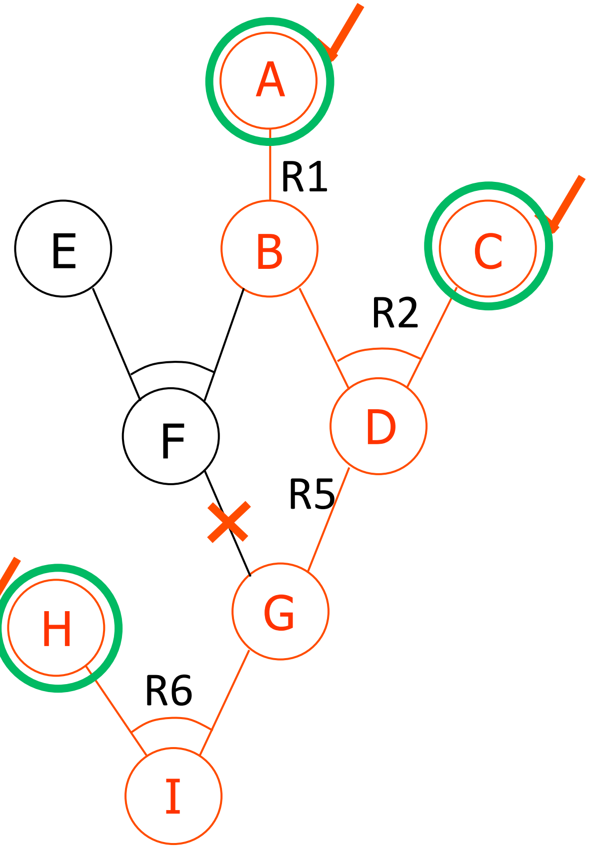
<u>In WM</u>: A,C and H



- B is not known to be true, but C is. We check B.
- Rule R1 supports B – post A as a goals

- R1: If A then B
- R2: If B and C then D
- R3: If B and E then F
- R4: If F then G
- R5: If D then G
- R6: If G and H then I

In WM: A,C and H



- A is known to be true.
- All supports for I found (R1, R2, R5, R6, A, C, H)
- I is proven

Backward Chaining in Jess

- Jess is a forward-chaining inference engine
- It provides mechanisms that "**simulates**" backward chaining – requires Jess programs to be programmed in a particular form

- Or we can program our own inference engine in Jess that does backward-chaining inference

Backward Chaining

- **When** do we need backward chaining?
  - Rules need to match particular supporting facts with LHS patterns
  - Other rules are fired **on-demand** to produce these supporting facts
  - E.g.: load additional facts from a database into WM

Dynamic situations, new facts keep on coming

Backward Chaining in Jess

- Rules **may wait for a particular fact** to be asserted
- Jess can help us in such a situation to actively fire other rules that would assert such a fact
  - Declare the corresponding **pattern** as "*backward chaining reactive*"
  - Jess will automatically assert a special pattern-specific fact
  - We can write rules that react to this special fact and assert the fact our rules waits for

Backward Chaining in Jess

- Declare deftemplates and ordered facts representing your goals for backward chaining
  - `(do-backward-chaining <fact/template name>)`
- Define rules with patterns ("goal patterns") on the LHS that react to these "reactive" facts or"goals"
- Jess treats these goal patterns in a particular way:
  - For each of these patterns, Jess **automatically** asserts so-called "goal-seeking" or "trigger" facts. These facts are derived from the goal patterns, but get the prefix "**need-**" added to their name.

Example

Jess needs a price fact to activate this rule

- Price check

```
(defrule price-check
    (do-price-check ?name)
    (price ?name ?price)
    =>
    (printout t "Price of " ?name " is " ?price crlf)

(reset)
(assert (do-price-check waffles))
```

```
(do-price-check waffles))
```

- Solution 1:
  - All prices are asserted into WM
- Solution 2:
  - We have to fetch the price from an external database and create price facts only on-demand – how to trigger the database access?

Example

```
(do-backward-chaining price)
```

Declare your fact for backward chaining,
We say: the fact “price” is now
“**backward-chaining reactive**”

```
(defrule price-check
    (do-price-check ?name)
    (price ?name ?price)
   =>
    (printout t "Price of " ?name " is " ?price crlf))
```

Example

```
(do-backward-chaining price)
```

Declare your fact for backward chaining

```
(defrule price-check
    (do-price-check ?name)
    (price ?name ?price)
    =>
    (printout t "Price of " ?name " is " ?price crlf))
```

Jess asserts a fact with prefix **"need-"**

```
(need-price ?name ?price)
```

"Trigger fact"

```
WM:
f-0    (MAIN::initial-fact)
f-1    (MAIN::do-price-check Fred)
f-2    (MAIN::need-price Fred nil)
f-3    (MAIN::do-price-check Ann)
f-4    (MAIN::need-price Ann nil)
f-5    (MAIN::do-price-check Mary)
f-6    (MAIN::need-price Mary nil)
```

Price set to nil, because we don't know it yet

Example

Declare your fact for backward chaining

```
(do-backward-chaining price)
```

```
(defrule price-check
    (do-price-check ?name)
    (price ?name ?price)
    =>
    (printout t "Price of " ?name " is " ?price crlf))
```

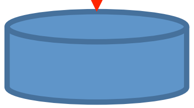
Jess asserts a fact with prefix **"need-"**

```
(need-price ?name ?price)
```

"Trigger fact"

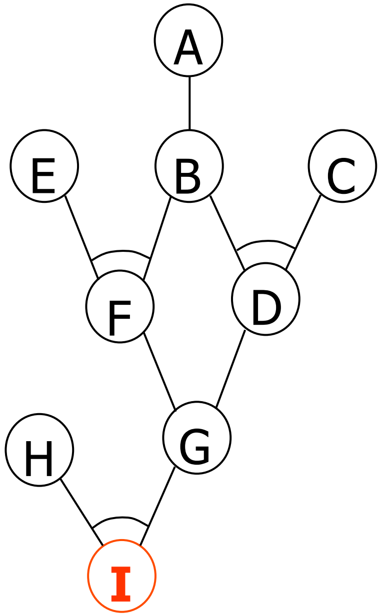
We can write rules that react to these facts

```
(defrule query-database
    (need-price ?name ?)
    =>
    (assert
        (price ?name (queryDB ?name)))
```



We assume to have implemented a function "queryDB".

Example




```
(do-backward-chaining goal)
(set-strategy breadth)

(defrule r1  (goal A) => (assert (goal B)))
(defrule r2  (goal B)(goal C) => (assert (goal D)))
(defrule r3  (goal B)(goal E) => (assert (goal F)))
(defrule r4  (goal F) => (assert (goal G)))
(defrule r5  (goal D) => (assert (goal G)))
(defrule r6  (goal G)(goal H) => (assert (goal I)))

(defrule r6-back (need-goal I) => (assert (proof G)(proof H)))
(defrule r5-back (need-goal G) => (assert (proof D)(proof F)))
(defrule r4-back (need-goal D) => (assert (proof B)(proof C)))
(defrule r3-back (need-goal F) => (assert (proof B)(proof E)))
(defrule r2-back (need-goal F) => (assert (proof B)(proof E)))
(defrule r1-back (need-goal B) => (assert (proof A)))


(defrule r7-proof
    (proof ?x)
    (goal ?x)
    =>
    (printout t "Achieved goal " ?x crlf)
)
```

```
(deffacts initialdata
   (goal A)(goal C)(goal H)
)
(reset)
(assert (proof I))
(facts)
(run)
(facts)
```

Example

```
f-0    (MAIN::initial-fact)
f-1    (MAIN::need-goal A)
f-2    (MAIN::need-goal B)
f-3    (MAIN::need-goal F)
f-4    (MAIN::need-goal D)
f-5    (MAIN::need-goal G)
f-6    (MAIN::goal A)
f-7    (MAIN::goal C)
f-8    (MAIN::goal H)
f-9    (MAIN::proof I)
f-10    (MAIN::need-goal I)
For a total of 11 facts in module MAIN.
Achieved goal A
Achieved goal C
Achieved goal B
Achieved goal H
Achieved goal D
Achieved goal G
Achieved goal I
```

```
f-0    (MAIN::initial-fact)
f-1    (MAIN::need-goal A)
f-2    (MAIN::need-goal B)
f-3    (MAIN::need-goal F)
f-4    (MAIN::need-goal D)
f-5    (MAIN::need-goal G)
f-6    (MAIN::goal A)
f-7    (MAIN::goal C)
f-8    (MAIN::goal H)
f-9    (MAIN::proof I)
f-10    (MAIN::need-goal I)
f-11    (MAIN::proof A)
f-12    (MAIN::proof B)
f-13    (MAIN::proof E)
f-14    (MAIN::need-goal E)
f-15    (MAIN::proof C)
f-16    (MAIN::proof D)
f-17    (MAIN::proof F)
f-18    (MAIN::goal B)
f-19    (MAIN::proof G)
f-20    (MAIN::proof H)
f-21    (MAIN::goal D)
f-22    (MAIN::goal G)
f-23    (MAIN::goal I)
```

Summary

- Jess Backward chaining

  …

- Question?

Refraction
Avoid Re-activation of a Rule

- Refraction is the act of preventing a rule from re-activating itself
- Declaration in deftemplate

```
(deftemplate person
    (declare (slot-specific TRUE))
    (slot name)
    (slot salary))
```

- Declaration in rule:

```
(defrule increment
    (declare (no-loop TRUE))
    ?f <- (person
            (name   ?name)
            (salary ?salary))
    =>
    (modify ?f (salary (+ ?salary 10)))
)
```