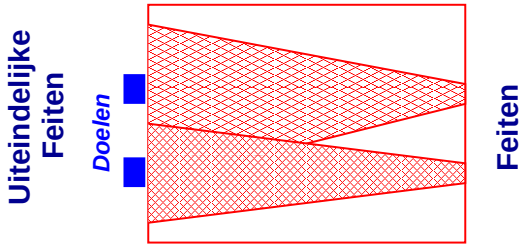
Bottom-up Inferentie

Data-gestuurde inferentievorm:



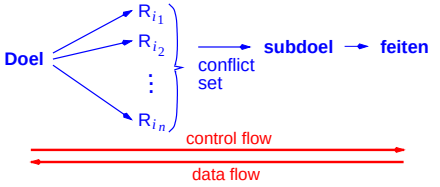
Voorbeeld:
Initiële feitenverzameling: $F = \{A\}$

Regelverzameling:
$\mathcal{R} = \{R_1$: **if** B **then** G **fi**, R_2 : **if** G **then** C **fi**, R_3 : **if** A **then** B **fi**$\}$

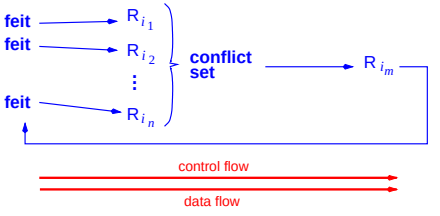
Na afloop: $F' = \{A, B, G, C\}$

Vergelijking

- Top-down inferentie:
  selectie regels *match* (sub)doel – conclusie



- Bottom-up inferentie:
  match feiten – condities



Basisalgoritme bottom-up inferentie

```
proc Infer(rule-base, fact-set)
  rules ← Select(rule-base, fact-set);
  while rules ≠ ∅ do
    rule ← ResolveConficts(rules);
    Apply(rule);
    rules ← Select(rule-base, fact-set)
  od
end
```

- alle matchende regels worden geselecteerd
- één regel wordt via conflictresolutie uitgekozen en toegepast
- globale sturing wordt wel recognise-act cycle genoemd
- zeer inefficiënt (vandaar Rete-algoritme)

Conflictresolutie (voorbeeld)

Voorbeeld:

Initiële feitenverzameling:

$$F = \{x = A, y = B, z = C, w = D\}$$

Regelverzameling (rule-base) $\mathcal{R}$:
$\mathcal{R} = \{R_1$: **if** $\text{same}(x, A)$ **and** $\text{same}(y, B)$ **then** $\text{add}(u, C)$ **fi**,
R_2 : **if** $\text{same}(x, B)$ **then** $\text{add}(u, F)$ **fi**,
R_3 : **if** $\text{same}(z, C)$ **and** $\text{same}(w, D)$ **then** $\text{add}(u, E)$ **fi**$\}$

Conflict set $\text{CS} = \{R_1, R_3\}$

Er zijn diverse conflictresolutiestrategieën bekend

Vormen van conflictresolutie

Conflictresolutie: kies één regel uit de *confict-set*

Voorbeelden:

- Prioriteit: expliciete specificatie van volgorde
- Forward-chaining: volgorde specificatie aanhouden
- Specificiteit: 'specifieke' regel wordt verkozen boven 'algemene' regel
- Recentheid: regel die slaagt met 'recente' feiten wordt verkozen boven regels die slagen met minder 'recente' feiten

Specificiteit

- R *is specifieker dan* R' *als* R *tenminste dezelfde condities bevat als* R' *(en eventueel meer)*. De *meest specifieke* regel wordt als eerste uitgevoerd

  Voorbeeld:

  R_1 : **if** same(prog-taal, symboolverwerking)
  then add(taal, AI) **fi**

  R_2 : **if** same(prog-taal, symboolverwerking)
  and same(syntax, haakjes)
  then add(taal, Lisp) **fi**

- Voordeel: kennisbank uitbreidbaar (specifieke regels vervangen automatisch algemenere regels)

Locale variabelen

- Vergelijk met *universeel gekwantificeerde variabelen* in predicatenlogica

  Voorbeeld:

  if
  same(persoon, inkomen, <x>) **and**
  lessthan(persoon, vaste-lasten, <x>)
  then
  add(persoon, lening, toegestaan)
  fi

- Regelinstantiatie: regel met constant(en) gesubstitueerd
- Voordeel: verhoogde expressiviteit

Voorbeeld

Gegeven regelverzameling $\mathcal{R}$:

R_1 : **if** same(x, A) **or** same(y, C)
then add(z, A) **fi**
R_2 : **if** same(x, B)
then add(u, D) **fi**
R_3 : **if** same$(x,$ <t>$)$ **and** same(u, D)
then remove(2) **fi**

(remove(2): feit dat matcht met tweede conditie wordt verwijderd) en feitenverzameling

$$F = \{x = \{A, C\}, y = C, u = D\}$$

De conflict-set bevat dan de volgende regelinstanties:
$(R_1, \{x = A\})$, $(R_1, \{y = C\})$,
$(R_3, \{x = A, u = D\})$, $(R_3, \{x = C, u = D\})$

Conflictresolutie o.g.v. recentheid

Feiten worden voorzien van een aanduiding van volgorde van opname in de feitenverzameling: time-tag

Notatie:
Een feit is een uitspraak van de volgende vorm: $t : x^s = C$, met $t \in \mathbb{Z}^+$, een time-tag, x^s een eenwaardige variabele, en C een constante

Feitenverzameling: $F = \{t_1 : x_1^s = C_1, \ldots, t_n : x_n^s = C_n\}$

Voorbeeld:
$F = \{1 : x = A,$
$2 : x = B,$ (tweede waarde anders,
$3 : y = C,$ maar niet inconsistent)
$4 : y = C\}$ (tweede waarde gelijk)

Conflictresolutie o.g.v. recentheid

- met elke productiegel die *slaagt*, worden de matchende feiten geassocieerd
- regelinstantie: productieregel met matchende feiten, die voorzien zijn van time-tags
- conflict-set wordt gesorteerd o.b.v. time-tags van matchende feiten
- de regelinstantie die boven in de ordening uitkomt, wordt gekozen, òf
- als meer dan één regelinstantie boven in de ordening uitkomt, dan wordt hieruit, door een tweede conflictresolutiestrategie, of op arbitraire gronden, één instantie gekozen

Voorbeeld

Gegeven is de regelverzameling $\mathcal{R}$:
R_1 : **if** $\mathrm{same}(x, A)$ **and** $\mathrm{same}(y, B)$
then $\mathrm{add}(z, A)$ **fi**
$\vdots$
met feitenverzameling

$$F = \{1 : x = A, 2 : x = A, 3 : y = B, 4 : y = B\}$$

Gecreëerde regelinstanties (conflict-set):
$(R_1, \{1 : x = A, 3 : y = B\})$
$(R_1, \{1 : x = A, 4 : y = B\})$
$(R_1, \{2 : x = A, 3 : y = B\})$
$(R_1, \{2 : x = A, 4 : y = B\})$

Ordening van regelinstanties

Ordeningsrelatie $R \geq R'$, met $R, R' \in \mathrm{CS}$:

- time-tags per regelinstantie in afnemende volgorde sorteren
- aanvullen met net zoveel nullen, tot aantal time-tags voor alle regelinstanties gelijk is
- lexicografische ordening van regels op basis van time-tag rijtjes

Opmerking: $\geq$ definieert een *totale* ordening

Voorbeeld

$$
\begin{aligned}
I_1 &= (R_1, \{6 : x = A\}) \\
I_2 &= (R_2, \{2 : y = B, 4 : z = C\}) \\
I_3 &= (R_3, \{1 : x = B, 3 : w = D, 5 : v = E\})
\end{aligned}
$$

Resultaat:

$R_1 : 6$	0	0
$R_2 : 4$	2	0
$R_3 : 5$	3	1

$\Rightarrow I_1 > I_3 > I_2$ (d.w.z. $I_1 \geq I_3$, maar $I_1 \neq I_3$)

Aanpassing bottom-up inferentie

```
proc Infer(rule-base, fact-set)
  applied-instances ← ∅;
  instances ← Select(rule-base, applied-instances, fact-set);
  while instances ≠ ∅ do
    instance ← ResolveConficts(instances);
    Apply(instance);
    applied-instances ← applied-instances ∪{instance};
    instances ← Select(rule-base, applied-instances,
                       fact-set)
  od
end
```

Elke instantie wordt ten hoogste eenmaal toegepast (een regel kan meer dan eenmaal worden geïnstantieerd)

Niet-monotoon redeneren

In productiesystemen met bottom-up inferentie wordt vaak van de actie `remove` gebruik gemaakt

Voorbeeld:
Initiële feitenverzameling $F = \{1 : y = B, 2 : x = A\}$, met regelverzameling $\mathcal{R}$:
R_1 : **if** $\text{same}(x, A)$ **then** $\text{remove}(1)$ **fi**
R_2 : **if** $\text{same}(y, B)$ **then** $\text{add}(z, C)$ **fi**

(remove(1): feit dat matcht met eerste conditie wordt verwijderd)

Inferentie:

(1) $I_1 = (R_1, \{2 : x = A\}), I_2 = (R_2, \{1 : y = B\})$

(2) $F' = \{1 : y = B\}$

(3) $F'' = \{1 : y = B, 3 : z = C\}$; stop

Oneindig redeneren

Productiesystemen met bottom-up inferentie kunnen oneindig blijven doorredeneren

Voorbeeld:
Initiële feitenverzameling $F = \{1 : x = A\}$
met regelverzameling $\mathcal{R}$:
R_1 : **if** $\text{same}(x, A)$ **then** $\text{add}(y, B)$ **fi**
R_2 : **if** $\text{same}(y, B)$ **then** $\text{add}(y, B)$ **fi**

Inferentie:

(1) $I_1 = (R_1, \{1 : x = A\})$

(2) Toepassen I_1: $F' = \{1 : x = A, 2 : y = B\}$

(3) $I_2 = (R_1, \{1 : x = A\}), I_3 = (R_2, \{2 : y = B\})$; I_2 valt af, want $I_1 = I_2$

(4) Toepassen van I_3: $F'' = \{1 : x = A, 2 : y = B, 3 : y = B\}$
...

Voorbeelden van systemen

- CLIPS: acroniem voor C Language Integrated Production System
  - Ontwikkeld door Lyndon B. Johnson Space Center van NASA
  - Hybride taal:
    - productieregels
    - object-georiënteerd programmeren
    - functioneel/procedureel programmeren
  - http://clipsrules.sourceforge.net/
- OPS5 (Carnegie-Mellon University) http://www.pcai.com/web/ai_info/pcai_ops.html
- JESS: Java-versie van regelgebaseerde deel CLIPS http://herzberg.ca.sandia.gov/

CLIPS voorbeeld

- **Object-declaratie = *deftemplate construct***

```
(deftemplate verhuisd
  (slot naam)
  (slot naar-adres))

(deftemplate persoon
  (slot naam (type STRING))
  (slot leeftijd (type INTEGER)))

(deftemplate nieuw-adres
  (slot adres))
```

- **Productieregels:**

```
(defrule verhuizing
  (verhuisd (naam ?naam)(naar-adres ?adres))
  (persoon (naam ?naam))
  =>
  (assert (nieuw-adres (adres ?adres))))
```


Feiten

Initiële feitenverzameling:

```
f-1 (persoon (naam Jan))
f-2 (persoon (naam Marco))
f-3 (verhuisd (naam Jan)
      (naar-adres Catharijne-singel-106))
f-4 (verhuisd (naam Marco)
      (naar-adres Neude-24))
```

```
(defrule verhuizing
  (verhuisd (naam ?naam)(naar-adres ?adres))
  (persoon (naam ?naam))
  =>
  (assert (nieuw-adres (adres ?adres))))
```

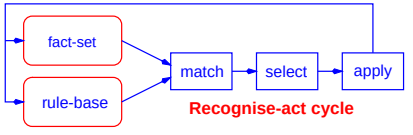
Na uitvoeren regel:

```
                ...
f-5 (nieuw-adres (adres Neude-24))
f-6 (nieuw-adres (adres Catharijne-singel-106))
```


Bottom-up inferentie

proc Infer(fact-set, rule-base)

applied-inst $\leftarrow \emptyset$;
inst $\leftarrow$ Select(rule-base, applied-inst, fact-set);
while $\text{inst} \neq \emptyset$ **do**
  instance $\leftarrow$ ResolveConflicts(inst);
  Apply(instance);
  applied-inst $\leftarrow$ applied-inst $\cup \{\text{instance}\}$;
  inst $\leftarrow$ Select(rule-base, applied-inst, fact-set)
od
end

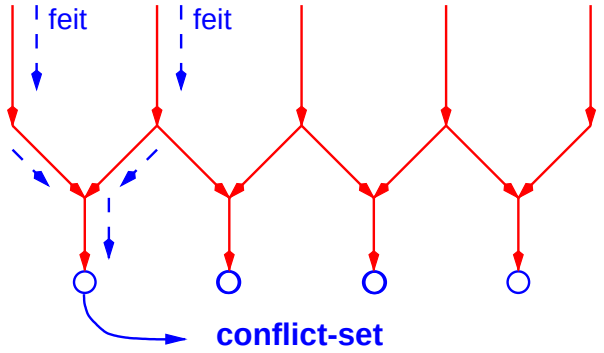


Rete-algoritme

- Efficiënte implementatie van bottom-up inferentie (Charles Forgy, Carnegie Mellon University)
- Motivering: bij meeste toepassingen wordt tot 90% van executietijd in beslag genomen door *selectie* en *match* operaties
- De feitenverzameling verandert slechts weinig tussen inferentiestappen: temporele redundantie
  $\Rightarrow$ inferentietoestand bewaren
- Rete-algoritme niet geschikt voor niet-temporeel redundante toepassingen (bijv. real-time systemen)

Basisideeën

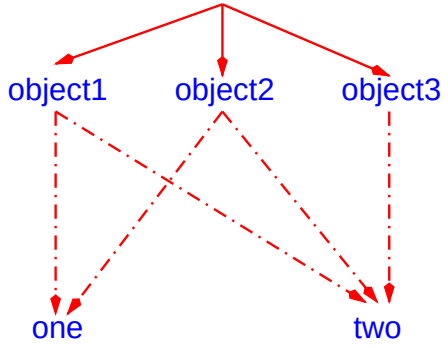
- Versnellen van matchen van feiten en condities: rete-graaf = compilatiegraaf van regels
- Bewaren van inferentietoestand door distributie van feiten over productieregels (rete-graaf)



Retegraaf: one-input nodes

Ingang van retegraaf (deftemplate names/ objectknopen/klassenknopen)

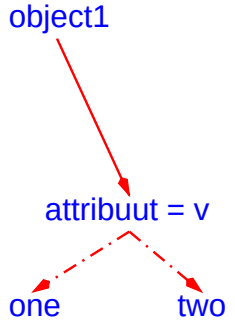
```
(defrule one
  (object1 ...)
  (object2 ...)
  =>
  ... )
(defrule two
  (object1 ...)
  (object2 ...)
  (object3 ...)
  =>
  ... )
```



Retegraaf: one-input nodes (vervolg)

Testen op attribuut(slot)waarden in een conditie

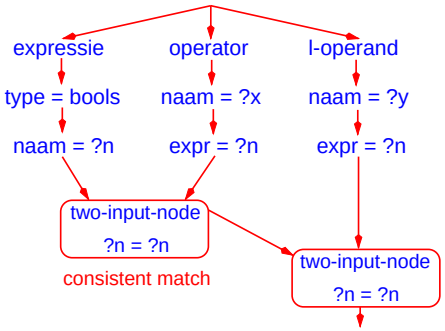
```
(defrule one
  (object1 (attribuut v))
           ⋮
  =>
  ...  )

(defrule two
  (object1 (attribuut v))
           ⋮
  =>
  ...  )
```



Retegraaf: two-input nodes

Conjuncties èn *negaties* van condities worden afgebeeld op two-input nodes

```
(defrule interpret
  (expressie (type bools) (naam ?n))
  (operator (naam ?x) (expr ?n))
  (l-operand (naam ?y) (expr ?n))
  =>
  ... )
```



Distributie van feitenverzameling

De retegraaf wordt uitgebreid met *geheugenknopen*:

- Met laatste uitgaande pijl van een rij one-input nodes wordt een α-geheugen geassocieerd.
  Deze bevat feiten die voldoen aan de tests van de voorgaande one-input nodes

- Met de uitgaande pijl van een two-input node wordt een β-geheugen geassocieerd.
  Deze bevat feiten die aan de tests van de twee inkomende pijlen, samen met de consistent matches, voldoen

Tokens

Een *token* is een verzameling feiten, samen met een indicatie of een feit *toegevoegd* is of *verwijderd*:
$\langle$ <*label*> <*feit*>$^+$ $\rangle$, met <*label*> ::=

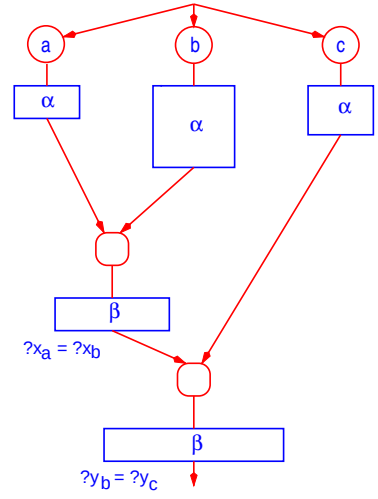
+ feiten zijn toegevoegd
- feiten zijn verwijderd

Manipulatie van tokens:

- Een token wordt aan een α-geheugen toegevoegd, als aan alle tests voorafgaande aan de inkomende pijl voldaan is
- Als de tokens van de twee inkomende pijlen van een two-input node $\langle$<*label*> $L\rangle$, resp. $\langle$<*label*> $R\rangle$, is, en de matches zijn consistent, dan wordt: $\langle$<*label*> $LR\rangle$ aan het β-geheugen toegevoegd

Voorbeeld retegraaf

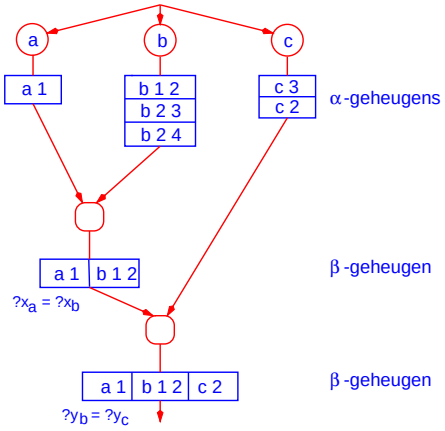
```
(defrule vb
  (a ?x)
  (b ?x ?y)
  (c ?y)
  =>
  ... )
```



Vullen retegraaf

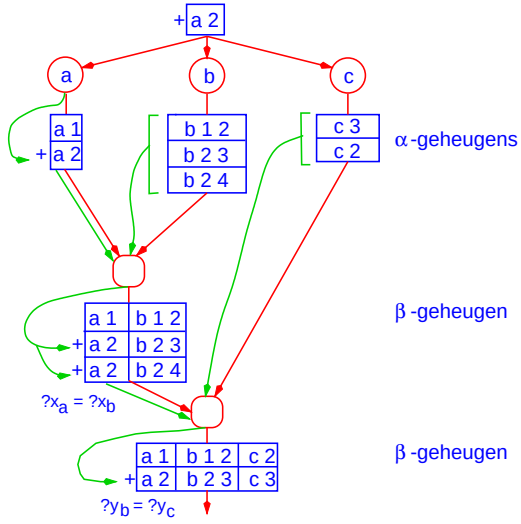
```
(defrule vb
  (a ?x)
  (b ?x ?y)
  (c ?y)
  =>
  ... )
```

```
(a 1)
(b 1 2)
(b 2 3)
(b 2 4)
(c 3)
(c 2)
```



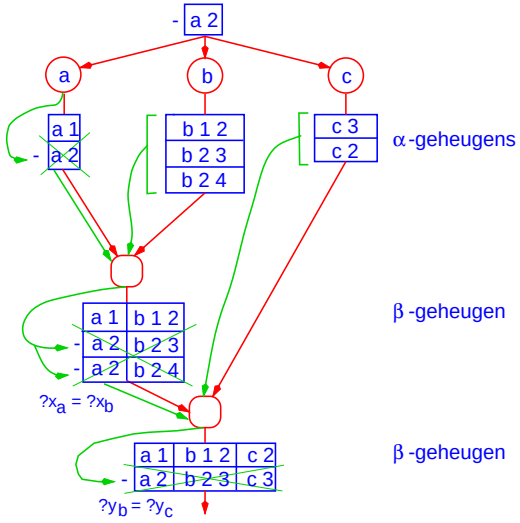
Toevoegen van tokens

```
(defrule vb
  (a ?x)
  (b ?x ?y)
  (c ?y)
  =>
  ... )
```



Verwijderen van tokens

```
(defrule vb
  (a ?x)
  (b ?x ?y)
  (c ?y)
  =>
  ... )
```



Voordelen van rete-algoritme

- Gegevens persisteren in de retegraaf tussen inferentiecycli
- ‘consistente match’ wordt slechts eenmaal bepaald
- structureel gelijke conditie-elementen worden slechts eenmaal gerepresenteerd
- eenvoudig af te beelden op de architectuur van parallelle computers

Nadelen van rete-algoritme

- Een β-geheugen kan het volledig cartesisch product van de inhouden van de bijbehorende α-geheugens bevatten

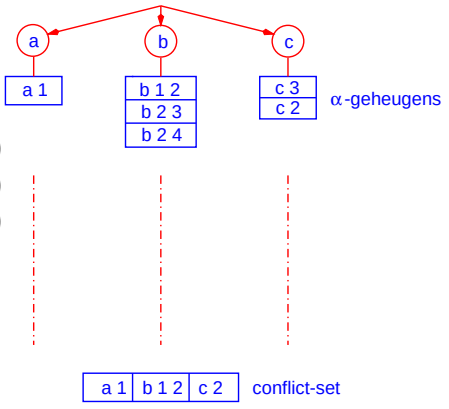
Dit nadeel heeft het *Treat*-algoritme niet:

- Ontwikeld door D.P. Miranker, *Treat: a new and efficient match algorithm for AI production systems*, London: Pitman, 1990.
- TREAT: “Temporally REdundant Associative Tree” algoritme, bedoeld voor:
  - temporeel redundante toepassingen, voor
  - parallelle machines met een boomvormig communicatienetwerk (Dado)
- Basisidee: geen β-geheugens, slechts de inhouden van de α-geheugens en de conflict-set worden bewaard

Voorbeeld treat-algoritme

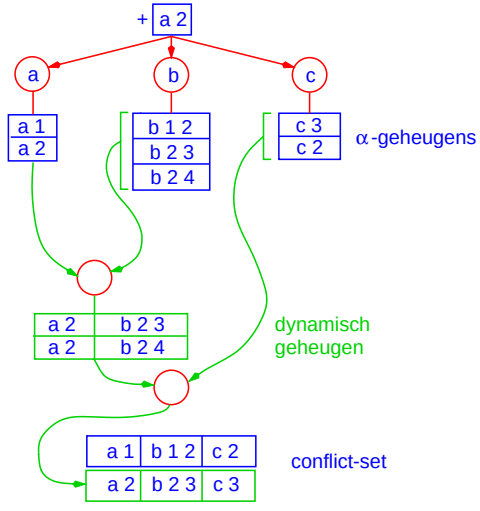
```
(defrule vb
  (a ?x)
  (b ?x ?y)
  (c ?y)
  =>
  ... )
```

```
(a 1)
(b 1 2)
(b 2 3)
(b 2 4)
(c 3)
(c 2)
```



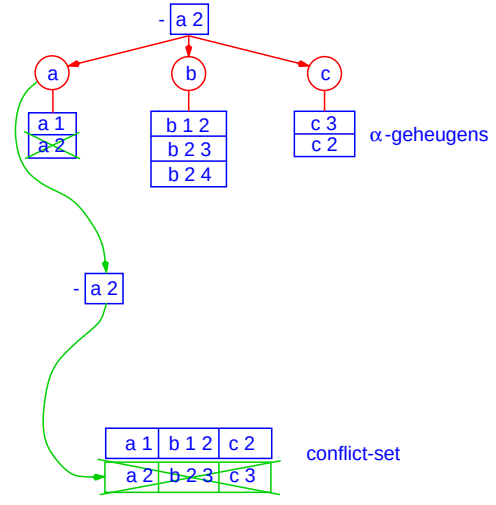
Toevoegen feit bij treat

```
(defrule vb
  (a ?x)
  (b ?x ?y)
  (c ?y)
  =>
  ... )
```



Verwijderen van feit bij treat

```
(defrule vb
  (a ?x)
  (b ?x ?y)
  (c ?y)
  =>
  ... )
```



Rete versus treat

- Toevoegen van nieuwe feiten: Rete minder vergelijkingen dan Treat (Treat: dynamisch geconstrueerd β-geheugen)
- Verwijderen van feiten: Rete meer vergelijkingen dan Treat
- Parallelle implementatie, zoals Dado: Treat sneller dan Rete

Opgaven

Zij gegeven: $F = \{1 : x = A, 2 : x = B, 3 : y = 4\}$ en regelverzameling $\mathcal{R}$:

R_1 : **if** $\text{same}(x, A)$ **and** $\text{same}(x, B)$
then $\text{add}(z, E)$ **fi**
R_2 : **if** $\text{same}(z, E)$ **and** $\text{same}(w, G)$
then $\text{add}(z, F)$ **fi**
R_3 : **if** $\text{lessthan}(y, 10)$ **and** $(\text{same}(x, A)$
or $\text{same}(x, B))$ **then** $\text{add}(w, G)$ **fi**

---

(1) Welke regelinstanties worden gecreëerd?

(2) Welke instantie wordt gekozen door confictresolutie op grond van recentheid?

(3) Welke feitenverzameling F' resulteert na toepassing van de gekozen regelinstantie?

Antwoorden

(1) Regelinstanties:

$$\begin{aligned} I_1 &= (R_1, \{1 : x = A, 2 : x = B\}) \\ I_2 &= (R_3, \{3 : y = 4, 1 : x = A\}) \\ I_3 &= (R_3, \{3 : y = 4, 2 : x = B\}) \end{aligned}$$

(2) Gekozen wordt I_3, want $I_3 > I_2 > I_1$

(3) Resulterende feitenverzameling: $F' = F \cup \{4 : w = G\}$

$F = \{1 : x = A, 2 : x = B, 3 : y = 4\}$; regelverzameling $\mathcal{R}$:
R_1 : **if** $\text{same}(x, A)$ **and** $\text{same}(x, B)$ **then** $\text{add}(z, E)$ **fi**
R_2 : **if** $\text{same}(z, E)$ **and** $\text{same}(w, G)$ **then** $\text{add}(z, F)$ **fi**
R_3 : **if** $\text{lessthan}(y, 10)$ **and** $\text{same}(x, A)$ **or** $\text{same}(x, B)$
then $\text{add}(w, G)$ **fi**