Intelligent Systems: Reasoning and Recognition

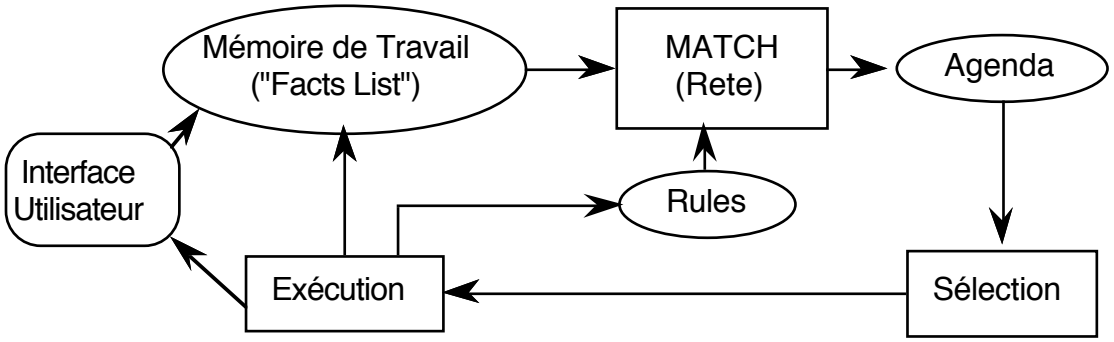
James L. Crowley

ENSIMAG 2 / MoSIG M1 Second Semester 2008/2009

Lesson 4 4 March 2009

Forward Chaining Rule Based Systems
Rule Matching and the Rete Algorithm

Architecture of a Rule Based Inference Engine



The Recognize-Act" cycle
The cycle has 3 phases

Correspondence	Matching facts to rule conditions to create "activations"
Selection	Selection of an activation (Rule + facts) for execution
Execution	Execution of the action part of the rule.

Every fact is identified by a unique index (or recency).
In each cycle, all facts are compared to all conditions of all rules.
This leads to a process with exponential algorithm complexity.

A rule and its matching facts are noted as an "activation"

Activations are stored in an "agenda". Each activtation indludes:
Rule name, rule priority (salience), the index of the fact that matches the each condition, and any variable assignments that occurred during matching.

By default the agenda is a stack (LIFO), however, this can be changed.

Rules in CLIPS

Syntax of "defrule"

(defrule <rule-name> [<comment>]
    [<declaration>] ; Rule Properties
    <conditional-element>* ; Left-Hand Side (LHS)
=>
    <action>*) ; Right-Hand Side (RHS)

If a rule with the same name does not exist, it is created. If it DOES exist, it is replaced.
There is no limit to the number of conditions or of actions in a rule.
When a rule is executed, its actions are executed sequentially in order that they are listed.

If a rule has no condition element, a default condition element matching "initial-fact" is used.

Condition elements can have many forms:

```
<conditional-element> ::= <pattern-CE> |
                          <assigned-pattern-CE> |
                          <not-CE> |
                          <and-CE> |
                          <or-CE> |
                          <logical-CE> |
                          <test-CE>
```

A condition element may be list of items or may be a structure defined by a template.

List : (<constant-1> ... <constant-n>)
Deftemplate :

```
(<deftemplate-name> (<slot-name-1> <constant-1>)
                                   •
                                   •
                                   •
                    (<slot-name-n> <constant-n>))
```

The condition may be a litteral, or may contain variables.

VARIABLES

There are 2 sorts of variables:

Index Variables: Can be assigned the indexes of fact that matches a condition element.
Attribute Variables: Can be assigned a variable that may span multiple conditions.
Attribute variables are used to match facts!

Index Variables

Index variables are used to record the index of a fact that matched a condition element. For example, to delete or modify the fact.

```
(defrule rule-A
   ?f <- (a)
=>
   (printout t "Retracting  " ?f  crlf)
   (retract ?f)
)

(deftemplate A (slot B (default 0)))

(defrule rule-A
   ?f <- (A (B 0))
=>
   (printout t "Changing  " ?f  crlf)
   (modify ?f (B 1))
)
```

Attribute Variables

Attribute values are used to recover attribute values from facts, and also used to match different facts.

Syntax:

?<NOM> - a variable for a single item. The name may not contain spaces or punctuations

$?<LISTE> - a variable for a list of items.

? - wildcard variable. Will match any item, but does not recover value

$? - Wildcard list. Will match any number (every number!) of successive items.

Examples :

```
(assert (a b c))
(assert (a b c d e f))
(assert (d e f))

(defrule "select second item"
  (a ?x  ?)
=>
(printout t "x = " $?x crlf)
)

(defrule "select a list"
  (a $?x)
=>
(printout t "the list is " $?x crlf)
)
```

The following is a technique to select EVERY item in a list:

```
(defrule "Print each item of a list"
  (a $? ?x $?)
=>
(printout t "x = " ?x crlf)
)
```

Example of the use of a variable :

```
((deftemplate person       ;  une relation pour un person
   "record pour une person"   ;  commentaire optionnel
   (slot famille           ; nom du person
      (type STRING)    ; Type chaine de caractères
      (default " Dupont")); Par défaut
   (slot prenom            ; nom du person
      (type STRING)    ; Type chaine de caractères
      (default "Pierre ")); Par défaut
   )

(defrule Find-same-name
   ?P1 <- (person (nom ?n1) (prenom ?b))
   ?P2 <- (person (nom ?n2) (prenom ?b))
=>
   (printout t ?B ?n1 " et   " ?B ?n2 "Ont le meme
prenom" crlf)
)
```

Syntax for CLIPS rules: Logical Constraints.

We can impose constraints on the facts that match condition elements.

There are two kinds of constraints: Boolean-connectors and predicates.

Boolean Connectors: CLIPS offers the classic logical operators: and, or, not

<v1> or <v2> - value v1 or value v2.

Variables are assigned values. These assignments can be subject to logical constraints.

example:

(?x & green | blue) - condition element is satisfied if ?x is green or ?x is blue.
(?x & ~rouge) - condition element is satisfied if ?x ≠ rouge

Examples of rules:

```
(defrule test3
  (couleur ?x & vert | bleu)
=>
  (assert (ok))
)

(assert (couleur vert))
(assert (couleur rouge))

(defrule feu-rouge
      (couleur ?x&~vert&~jaune)
=>
      (assert (il faut arreter))
    (printout t "arret" crlf)
)
```

Predicates

The assignments of values to variables can be conditioned on predicates:
In this case, the variable is followed by a ":"

(?x&:(<predicate> <<arguments>>) Satisfied if variable matched and predicate is true
(?x|: (<predicate> <<arguments>>) Satisfied if variable matched or predicate is true

(?x&~(<predicate> <<arguments>>) Satisfied if variable matched and predicate is not true

Predicates:

(numberp <arg>) - true if <arg> is a number
(stringp <arg>) - true if <arg> is a string
(wordp <arg>) - true if <arg> is a word

```
(defrule example-1
  (data ?x&:(numberp ?x))
  =>)

(defrule example-2
  (data ?x&~:(symbolp ?x))
  =>)

(defrule example-3
  (data ?x&:(numberp ?x)&:(oddp ?x))
  =>)

(defrule example-4
  (data ?y)
  (data ?x&:(> ?x ?y))
  =>)

(defrule example-5
  (data $?x&:(> (length$ ?x) 2))
  =>)
```

Functions

A condition element can be made dependent on the execution of a function.
This can include a predefined function or a user defined function.
Functions can be in the conditions, the actions or interpreted by the listners.

A function is executed by a "test".

Syntax : (test (<fonction> [<<args>>]))

There exist several classes of predefined functions.
Il existe plusieurs classes des fonctions prédefinit.

Logical Functions:

Function	Symbol
negation	not
conjunction	and
disjunction	or

Function for comparisons of values:

Function	Symbole	exemple
Numerical Equality	=	(test (= ?x ?y))
equivalence	eq	(test (eq ?nom ?mere))
numerical inequivalence	!=	(test (!= ?x ?y))
inequivalence	neq	(test (neq ?nom ?mere))
Superior	>	(test (> ?x ?y))
Superior or eq	>=	(test (>= ?x ?y))
inferior	<	(test (< ?x ?y))
inferior or eq	<=	(test (<= ?x ?y))

Arithmatic:

division	/	(test (< ?x (/ ?y 2)))
multiplication	*	(test (< ?x (* ?y 2)))
addition	+	(test (> (+ ?y 1) ?max))
subtraction	-	(test (< (- ?y 1) ?min))

Deffunctions

User defined functions are created with "deffunction"
a function can return a string, a word or a number.

Syntaxe :

```
(deffunction <name> [<comment>]
   (<regular-parameter>* [<wildcard-parameter>])
   <action>*)

<regular-parameter> ::= <single-field-variable>
<wildcard-parameter>::= <multifield-variable>
```

examples :

```
(deffunction ma-fonction (?x)
   (printout t "L'argument est " ?x crlf)
)

(ma-fonction fou)

(deffunction test (?a ?b)
       (+ ?a ?b) (* ?a ?b))

(test 3 2)


(deffunction distance (?x1 ?y1 ?x2 ?y2)
 (bind  ?dx (- ?x1 ?x2))
 (bind  ?dy (- ?y1 ?y2))
   (sqrt (+ (* ?dx ?dx) (* ?dy ?dy)))
)
```

Actions

In the action part, an external function may be exectured with:

(<fonction> <<args>>)

example :

```
(deffunction distance (?x1 ?y1 ?x2 ?y2)
  (bind  ?dx (- ?x1 ?x2))
  (bind  ?dy (- ?y1 ?y2))
  (sqrt (+ (* ?dx ?dx) (* ?dy ?dy)))
)

(defrule calcul-distance
    (point ?x1 ?y1)
    (point ?x2 ?y2)
=>
(assert
    (distance (distance ?x1 ?y1 ?x2 ?y2)))
)
```

User defined function can make use of the following:

bind - assign a value to a variable (bind ?x 1)

read, readline - read data from the tty.

exemple :

```
(defrule ask-user
    (person)
=>
    (printout t "Prenom? ")
    (bind ?prenom (read))
    (printout t "Nom de famille? ")
    (assert (person ?prenom =(read)))
)
```

Note that with bind, parantheses indicate a function to execute.

Les actions Prédefinis

1) assert creates a fact in the facts list.

Syntax : (assert (<<fait>>) [(<<faits>>)])

```
(defrule  j'existe
    (je pense)
=>
    (assert (j'existe!))
)
```

2) retract - removes a fact from the facts list

```
(defrule je-n'existe-pas
    ?moi <- (je ne pense pas)
=>
    (retract ?moi)
)
```

3) Str-assert asserts a string as a fact

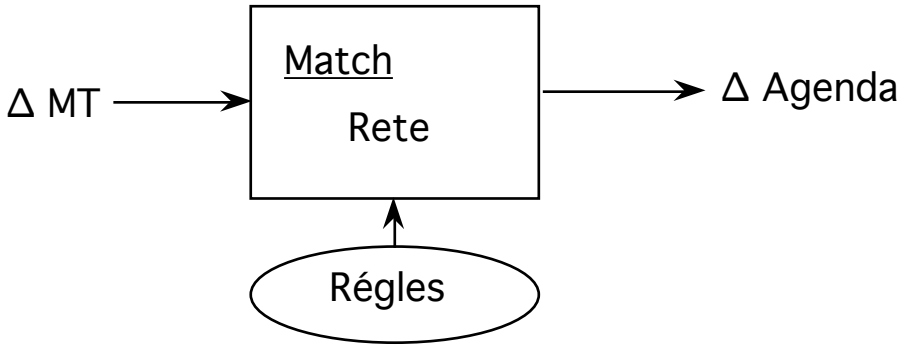
```
(defrule j-existe-je-pense
    (je pense)
=>
    (str-assert "Je pense que j'existe")
)
(facts)

f-0   (Je pense que j'existe)
```

4) Halt : Terminate execution.

L'Algorithme RETE

Appariement des Faits et Règles

Dans un système de productions, en principe, pour chaque condition de chaque règle, il faut parcourir la liste des faits.
Afin d'éviter le coût de calcul CLIPS (et OPS-5 et ART) utilise l'algorithme RETE.



RETE est une algorithme incrémentale d'unification.
De ce fait, RETE réduit fortement le temps d'exécution d'un système à règles.

L'algorithme RETE "évite" l'itération.

RETE évite l'itération. Il est incrémentale. Il fonction avec les modifications (assert, retract, modify) de la liste des faites (Δfaits).

RETE utilise un réseau de décision fait par la compilation des règles.
L'arbre de décision contient l'information de la mémoire de travail.

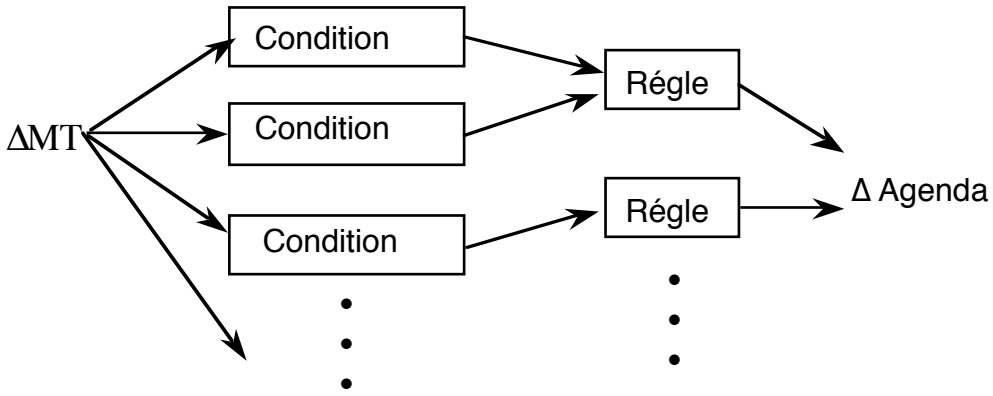
RETE est le mot latin pour Réseau.
Une modification de la Mémoire de Travail est propagée à travers un "réseau" pour engendrer des modifications dans l'agenda.
L'agenda est composé d'une liste des "activations".

Le réseau de décision

La partie condition des règles est composée de testes ("CONDITIONS").

```
(defrule   nom
        (condition-1)
        (condition-2)
=>
       (actions)
)
```

Chaque condition de chaque règle est compilé en filtre :



Les filtres de conditions sont indexés par

- Le "type" d'un template, ou
- La premier item d'une liste.

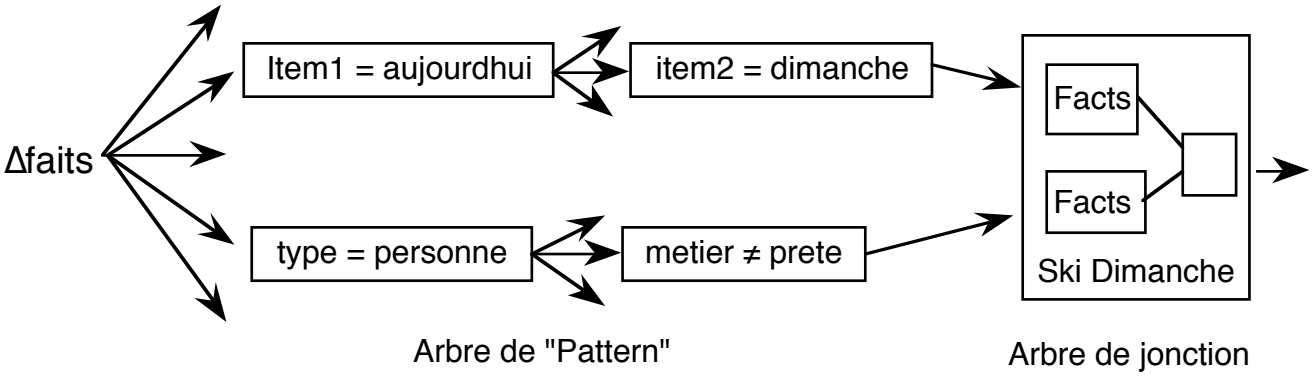
(Depuis clips 6, la première item d'une liste sert de "type" pour la liste.
D'où la nécessité de commencer les faits par un "symbole"

Ces filtres forme un arbre de patterns.

Exemple d'un arbre de décision de RETE,

```
(deftemplate personne
       (slot nom)
       (slot metier)
)

(defrule ski-dimanche
       (aujourdhui dimanche)
       (personne  (metier ?p:&~pretre))
=>
```

```
        (assert (fait du ski))
    )
```



L'arbre de jonctions : L'unification de faits

Pour chaque règle il y a un arbre de jonction. Il sert à

1) maintenir une liste des objets qui satisfont chaque condition de la règle.
2) assure que les attributs variables sont les mêmes.

À l'entre de l'arbre de patterns, il y a la liste de faits ayants satisfait les conditions (passée l'arbre de patterns).

A l'intérieur, il y a un arbre de tests pour unifier les faits.
Ces testes mettres en correspondance les variables d'attributs.

Exemple : :

```
(defrule exemple
    (personne nom ?x)
    (not (père-de ?x  ?y))
    =>
```

Consignes pour l'éfficacité de RETE

Principes pour l'organisation du parti « Condition » des règles. Ces principes sont une conséquent du fait que les conditions sont évaluées de l'haut vers le bas.

1) Placer la spécifique avant générale. Les règles avec le moins de variables et jokers en avant.
2) Placer les formes rares en avant. Chaque condition concerne une classe de fact. Placez en premier les conditions concernent les classes rares.
3) Tester les facts « volatile » en derniers.

Remarque sur la Complexité :

Pour le temps d'exécution d'un cycle :

Soit :

P : Nombre de règles
W : Nombre d'éléments dans les faits
C : Nombre de conditions dans une règle.

La complexité de calcul d'un cycle "recognize-act" est

Meilleur cas $O(\log_2 P)$

plus mauvais cas $O(P W^c)$

Dans les cas normals, la croissance de temps d'exécution est pratiquement linéaire avec le nombre de faits et de règles, mais avec un taux de croissance très faible. On peut, ainsi, avoir les systèmes avec les milliers de faits et règles.

L'Agenda

AGENDA : l'agenda est une liste des instances des règles associées avec les variables. Chaque activation est une association des faits et règles.

Réfraction: Une fois qu'un fait est associe à une règle est exécuté, l'association est éliminée de l'agenda.

Sélection (Résolution de Conflit) :

Principes de Sélection :

Réfraction : Une association de règles et faits peut être exécutée qu'une fois.

Recency : Les activations sont triées utilisant le plus grand indice de leurs faits.
Variation : MEA, Le fait satisfaisant la premiere condition détermine "recency" utilisé de trier l'activation .

Spécificité : Les activations sont triées au base de nombre de tests dans les conditions.

Par exemple :

```
(defrule example
   (item ?x ?y ?x)
   (test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))
   =>)
```

a specificity 2

Stratégies de Résolution de conflit :

CLIPS contient sept modes de "Sélection" (Stratégies de résolution de conflit)

1) "Depth Strategy" (par profondeur): Option par défaut.
L'agenda est trié par "saliency",
pour des valeurs de "saliency" égales,
l'agenda est une PILE d'activations. (LIFO)

2) "Breadth Strategy" (largeur d'abord).
L'agenda est trié par "saliency", pour une valeur de "saliency" l'agenda est une queue des activations. (FIFO)

3) LEX strategy (Lexographic). Pour compatibilité avec OPS-5.
L'agenda est une PILE, pas de saliency. Les activations sont triée par le "recency" des faits. Pour les activations le
plus récent, les activation sont triées par le nombre de conditions. (mélange de "depth" et "complexity").

4) MEA strategy (Means-Ends-Analysis) Pour compatibilité avec OPS-5 Les activations sont triées sur la base du recency de la première condition de la règle, puis par nombre de conditions

5) Complexity Strategy: Les règles avec le plus de conditions ont priorité.

6) Simplicity: Les règles avec le moins de conditions ont priorité.

7) Random: Aléatoire.

Le strategy "Depth" est recommandé.

Illustration de l'Agenda :

```
 (set-strategy depth)
 (get-strategy)

(defrule rule-A
      ?f <- (a)
=>
      (printout t "Rule A fires with  " ?f  crlf)
)

(defrule rule-B
      ?f <- (b)
=>
      (printout t "Rule B fires with  " ?f crlf)
)

(defrule rule-A-and-B
      ?f1 <- (a)
      ?f2 <- (b)
=>
      (printout t "Rule B fires with  A =" ?f1 " and B = " ?f2  crlf)
)

(assert (a))
(assert (a))
(assert (b))

(set-strategy depth)
(set-strategy breadth)
(set-strategy lex)
(set-strategy mea)
(set-strategy complexity)
(set-strategy simplicity)
(set-strategy random)
(set-strategy depth)
```

Salience :

Normalement l'agenda est une pile.
CLIPS fournie une méthode de sélection de priorité des règles. "Salience"
(sailliance)

```
(defrule exemple
        (declare (salience 99))
        (initial-fact)
    =>
        (printout "J'ai un salience de 99" crlf)
)
```

Le valeur de salience peut être entre –10 000 et 10 000.

Il y a tendance pour les débutants d'abuser le "salience".
Ce permet de commander l'ordre d'exécution des règles. Or, justement, si le système est bien conçu, on n'a pas besoins d'être concernées par l'ordre exact des règles, mais plutôt des blocs de règles.

Un système bien fait utilisera 3 ou 4 niveaux de salience.
Plus que 7 doit jamais être nécessaire.

En place de salience, il faut structurer les règles avec les éléments de contrôle.

Hiérarchie des Saliences :

En général dans un système expert, il y a quatre niveaux de salience.
Je les marque ici par les échelles de 100, mais les chiffres exacts sont arbitraires.

Niveau	Salience	
Contraints	300	;; Règles qui permettre une réduction des hypothèses.
Expert	200	;; Connaissance du domaine (inférer et trier les hypothèses)
Query	100	;; Interroger l'utilisateur
Contrôle	0	;; transitions de phases.

Cette hiérarchie dépend du problème et du régime de Contrôle.