Intelligent Systems: Reasoning and Recognition

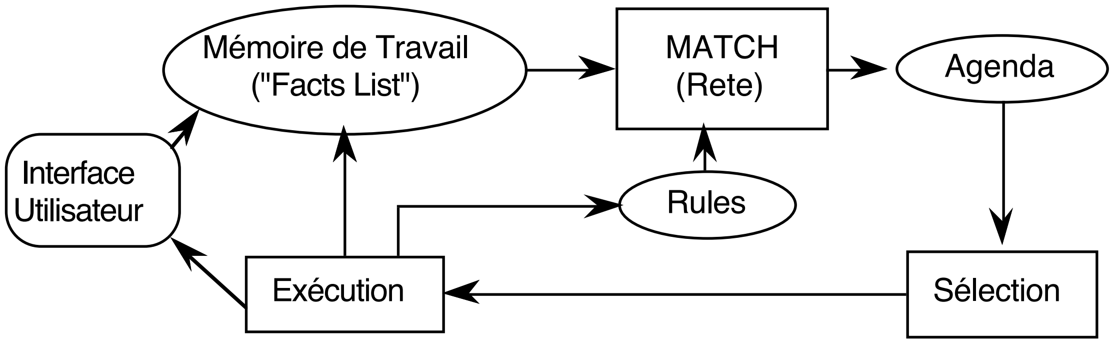
James L. Crowley

ENSIMAG 2 / MoSIG M1 Second Semester 2013/2014

Lesson 7 12 march 2014

CLIPS: RULE Syntax, Actions, The RETE Algorithm

Production System Architecture



The system implements an "inference engine" that operates as a 3 phase cycle:

The cycle is called the "recognize act" cycle.
The phases are:
MATCH: match facts in Short Term memory to rules
SELECT: Select the correspondence of facts and rules to execute
EXECUTE: Execute the action par of the rule.

Rules in CLIPS (continued)

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

Functions

A CE can depend on the result of a function.
There are a number of predefined functions in CLIPS.
Additional functions can be defined by the programmer.

In the Condition elements, a function is executed with the command (test)

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

There exist many predefined functions (see manual).

comparison functions

Function	Symbol	example
Equality	=	(test (= ?x ?y))
Equivalence	eq	(test (eq ?nom ?mere))
Numerical Inequivalence	!=	(test (!= ?x ?y))
Symbolic inquivalence	neq	(test (neq ?nom ?mere))
Greater than	>	(test (> ?x ?y))
Greater than or eq	>=	(test (>= ?x ?y))
Less than	<	(test (< ?x ?y))
Less than or equal	<=	(test (<= ?x ?y))

Arithmetic :

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

Deffunctions

The user may define his own functions with defunction.
A user defined function returns a value. This may be a string, symbol, number or any primitive.

Syntax:

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

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

examples :

```
(deffunction my-function (?x)
   (printout t "The argument is " ?x crlf)
)

(my-function 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)))
)
```

The ACTION part (RHS) of a rule

In the action part (or RHS) the rule contains a sequence of actions.
Any command recognized by the interpreter can be placed in the action part of a rule.
Each action enclosed in parentheses (<fonction> <<args>>*)
The first symbol in parentheses is interpreted as a function.

example :

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

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

New variables can be defined and assigned with bind: (bind ?x 0).
Values may be read from a file or from ttyin by read and readline.

example :

```
(defrule ask-user
   (person)
=>
   (printout t "first name? ")
   (bind ?surname (read))
   (printout t "Family name? ")
   (assert (person ?surname (read)))
)
```

System Actions

1) assert :   facts are created with "ASSERT"

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

```
(defrule I-Think-I-Exist
    (I think)
=>
    (assert-string "(I exist)")
)
```

2) retract - Facts are deleted with retract

```
(defrule I-dont-think-I-Exits
    ?me <- (I do not think)
=>
   (printout t "oops!" CRLF)
   (retract ?me)
)
```

3) Str-assert Assert a string

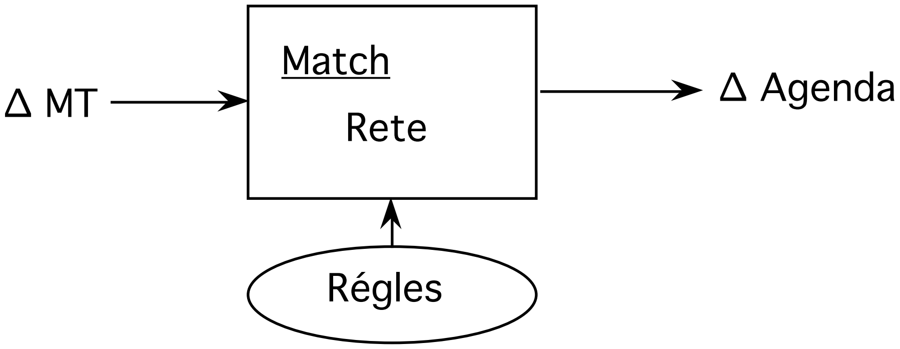
```
(defrule I-Think-I-Exist
    (I think)
=>
    (str-assert "I Think therefore I exist")
)

(facts)
```

4) Halt : Stop execution.

The RETE Algorithm

In a production system, in principle, each condition of each rule requires a complete scan of the working memory (facts list) during each cycle of execution. This can be very costly.

The RETE algorithm avoids this by providing incremental matching between facts and the LHS of rules.



RETE is an incremental matching algorithm. The word RETE is Latin for "network".
RETE operates by compiling the rules into a decision network.
The inputs to the algorithm are changes to working memory.
The outputs are changes to the agenda.

The working memory can only be changed by the commands assert, retract, modify or reset. Modify can be implemented as retract then assert. Reset clears all facts.

Changes in working memory filter through this decision network generate changes to the agenda.

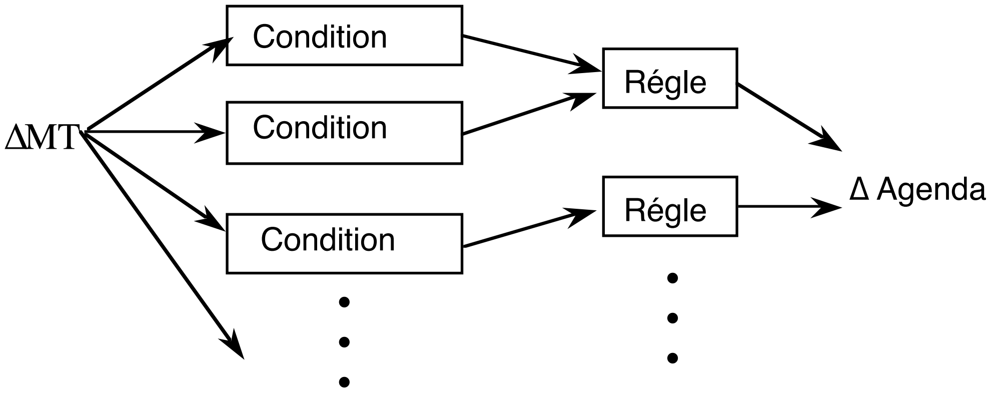
The RETE decision network

The condition (LHS) part of a rule is composed of a list of Condition Elements (CEs)

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

Each CE can be considered as a form of filter for a certain type of facts.
The type is the type defined by the template, or the first symbol of the fact.

Groups of CEs for the same type are grouped into a sub-network.

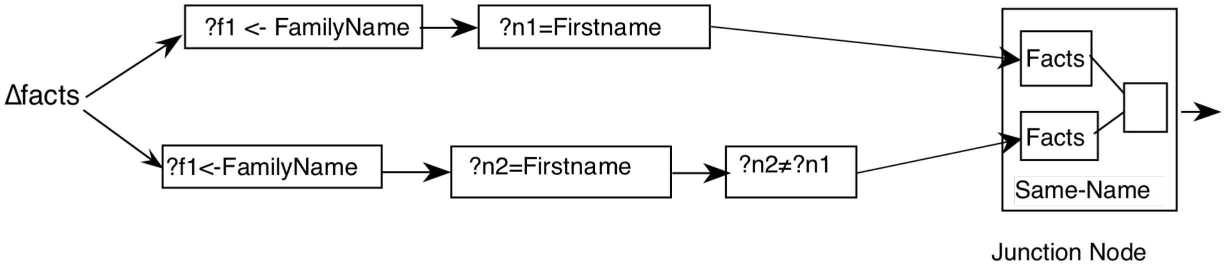


The network dispatches each change in working memory (facts) to the filter group for the "type" of the fact.

For example, consider :

```
(deftemplate person
	(slot family-name)
	(slot first-name)
)
```

```
(defrule Same-name
   ?P1 <- (person (family-name ?f)(first-name ?n1))
   ?P2 <- (person (family-name ?f)(first-name ?n2&~?n1))

=>
   (printout t  ?n1 " " ?f" and  " ?n2 " " ?f " have the
same family name" crlf)
)
```



For each rule, there is a filter branch for each CE. The filter branches meet at a Junction element.

Junction Elements: Unification of facts.

Junction elements have two roles:

1) to maintain the list of all fact indexes for all facts that satisfies each CE of the rule.
2) to match variables between CE's to produce lists of facts for which a variable is assigned the same value.

Each input to the junction maintains a list of facts that satisfied the CE.
Each time a list is changed, any variable assignments are compared to the variable assignments for all other CEs of the rule.

A list of indexes for matching facts is produced.

Example:

```
(deftemplate person (slot name) (slot father) (slot gender)

(defrule example
	(person (name ?father))
	(person (father ?father) (name ?child)
=>
	(printout ?father" is the father of  " ?child crlf )
)
```

CEs can be negative. Consider :

```
(defrule example
	(person (name ?n))
	(not (person (father ?n)))
	=>
	(printout t "?n"  has not children " crlf)
)
```

Efficient programming with RETE

The order of CE's in a rule can affect the efficiency of a rule base.
This is because the Join evaluates CE's in the order that they are declared.
Advice for more efficient code:

1) Place specific tests before more general tests. The more variables and wildcards in a CE, the lower it should go. Comparing variable bindings is expensive.
2) CEs that are least likely to match should be given first.
3) Volatile patterns (CEs that concern facts that are frequently modified, asserted or retracted, should be listed last.
4) Multifield and $? variables should be used carefully. They are more expensive because the bind zero or more fields.
5) Many short simple rules are better than a few complex rules.

Algorithmic Complexity of RETE:

Given: P: Number of rules
C: Average number of CEs in a rule
W: Number of facts

The algorithmic complexity of the recognize act cycle is:

Best case: O(Log(P))
Average Case O(PW)
Worst Case: $O(PW^c)$

The worst case happens when there are many variables to match.
For simple rule bases with few variable matches, computation and memory grow slightly faster than linear.

Programs with thousands of rules and tens of thousands of facts are practical.

The Agenda

The agenda is a list of activations of rules. It provides the rule name, index of the fact that matches each CE, and variable bindings. There are a number of different control regimes.

Control regimes following different principles.
A fundamental principle is "Refraction" .

Refraction: A unique activation is only executed once.
Activation is removed from the agenda on execution.

By default, the agenda acts as a stack (LIFO).
Other general principles include:

recency: Recent activations are given priority
MEA: A variation of recency where the index of the fact matching the FIRST CE determines the priority of the activation.
specificity: Rules with more CEs are given priority.

For example:

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

has specificity 2

OPS 5 provided 2 control regimes: LEX and MEA.
CLIPS has 7.

1) "Depth Strategy" - Agenda acts as a list of stacks (LIFO).
There is a separate stack for each salience.

2) "Breadth Strategy" - Agenda acts as a list queue (FIFO) with a separate stack for each salience

3) LEX strategy (Lexographic). (Compatibility with OPS). The agenda is a list of sorted activations. Activations for each saliency are sorted by Recency and then by specificity.

4) MEA strategy (Means-Ends-Analysis). (Compatibility with OPS). Activations for each saliency are sorted based on Fact-Index of the FIRST CE, then by specificity.

5) Complexity Strategy: Rules are sorted by Specificity with most complex rules given priority.

6) Simplicity: Rules are sorted by specificity with simplest rules given priority.

7) Random: using a random seed determined at start of execution.

Depth strategy is recommended (Agenda acts as a stack).

Examples of Strategies

```
 (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

The salience property for a rule determines its priority.
Salient rules are given higher priority in the agenda.

Salience is "declared" in the [<declaration>] part of the LHS, before the CE's

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

(declare (salience S)) where -10 000 < S < 10 000
by default S is 0.

example:

```
(defrule example
            (declare (salience 999))
            (initial-fact)
      =>
            (printout "I am an important rule! Salience= 999" crlf)
)
```

There is a tendency for beginners to abuse salience in order to force the order of rule execution. Don't! Rules should be structured with contexts.
If the system is well constructed, rule execution order is not important and only a few saliencies are needed.

A well-constructed program should need only 3 or 4 salience. At most 7 may be needed.

Salience Hierarchy:

Different styles of programs can require different hierarchies of salience.
A good practice is to declare the hierarchy in advance, using multiples of 100.
An example is the following:

Level	Salience	
Constraints	300	;; Rules that eliminate hypotheses
Expertise	200	;; Domain knowledge
Query	100	;; Rules that interrogate the user
Control	0	;; Context transitions