Intelligent Systems: Reasoning and Recognition

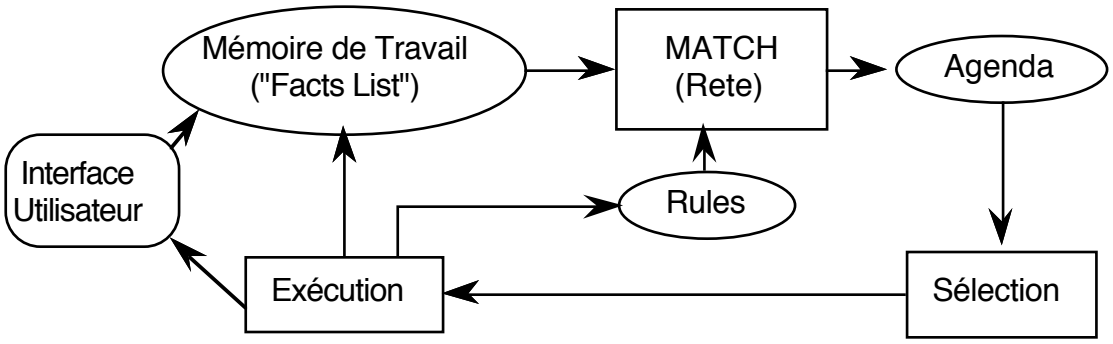
James L. Crowley

ENSIMAG 2 / MoSIG M1 Second Semester 2010/2011

Lesson 7 23 February 2011

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

CLIPS rules allow programming of reactive knowledge.
Rules are defined by the "defrule" command.

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

Rule Syntax: Constraints

Variable assignment and matching in conditions can be "constrained" by constraints.

There are two classes of constraints: "Logic Constraints" and Predicate Functions

Logic Constraints are composed using"&", "|", "~"

"&"- AND - Conjunctive constraint
"|" - OR - Disjunctive Constraint
"~" - NOT - Negation

example :
(?x & green | blue) - ?x must be green or blue for the condition to match
(?x & ~rouge) - ?x cannot match red.

```
(defrule Stop-At-Light
  (color ?x & red | yellow)
=>
  (assert (STOP))
  (printout t "STOP! the light is " ?x crlf)
)
 (assert (color red))
```

Predicates

Predicates provide functions for defining constraints.
For Predicate functions, the variable is followed by ":".

(?x&:(<predicate> <<arguments>>)	The condition is satisfied if 1) a value is assigned to ?x , and 2) the predicate is true for arguments
(?x|: (<predicate> <<arguments>>)	The condition is satisfied if 1) a value is assigned to ?x , or 2) the predicate is true for arguments
(?x&~(<predicate> <<arguments>>)	The condition is satisfied if 1) a value is assigned to ?x , and 2) the predicate is false for arguments

There are many predefined predicates. For example.

(numberp <arg>) - true if <arg> is a PRIMITIVE of type NUMBER
(stringp <arg>) - true if <arg> is a PRIMITIVE of type STRING
(wordp <arg>) - true if <arg> is a PRIMITIVE of type WORD

Additional functions can be found in the manual

```
(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 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

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

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>>)

functions are executed by being first symbol in parentheses.

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

2) retract - Facts are deleted with retract

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

3) Str-assert Assert a string

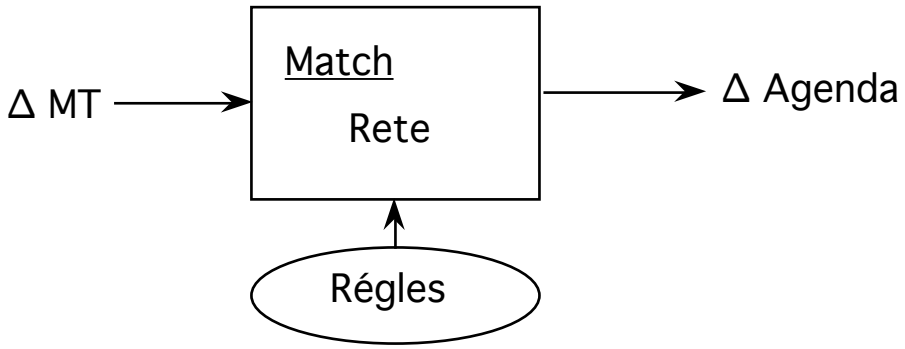
```
(defrule j-existe-je-pense
     (je pense)
=>
     (str-assert "Je pense que j'existe")
)
(facts)

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

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.

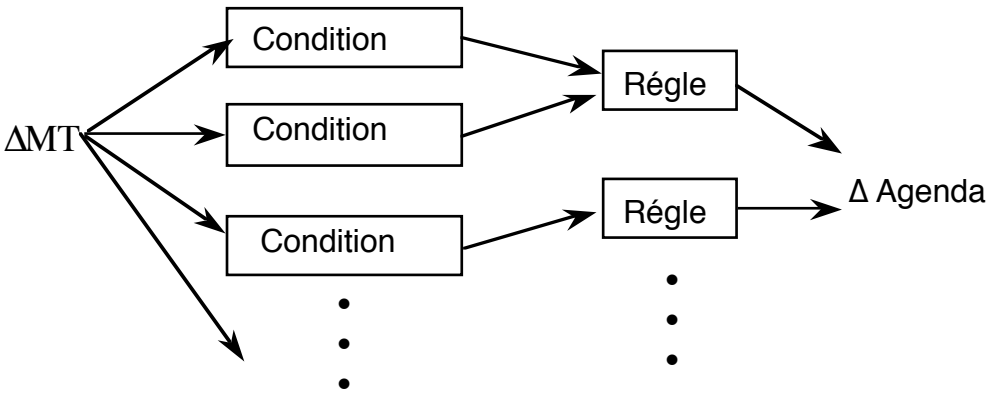
The RETE decision network

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.

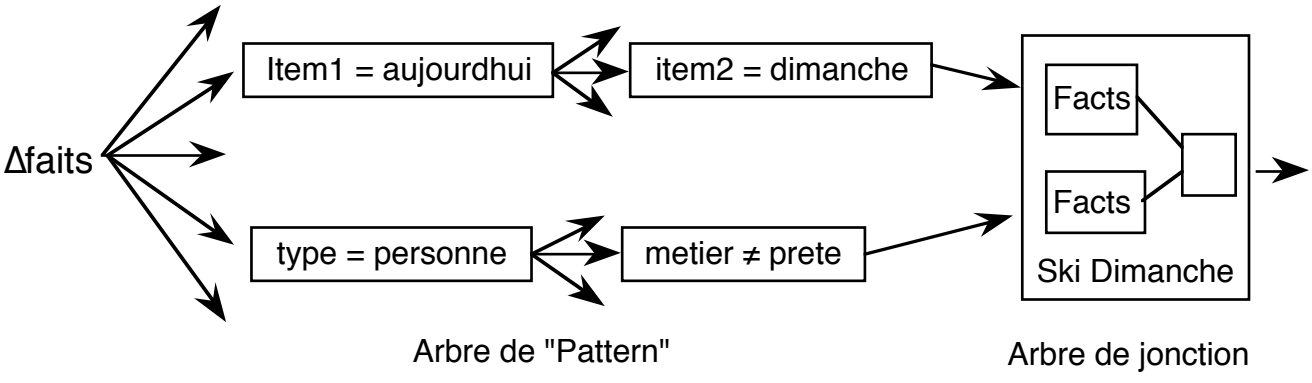
Groups of CEs for the same type are grouped into a sub-network.



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 name)
      (slot profession)
)
(assert (person profession sunday))

(defrule ski-on-sunday
      (today sunday)
      (person  (profession ?p:&~priest))
=>
      (assert (go skiing))
)
```



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.