

# Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 / MoSIG M1

Second Semester 2013/2014

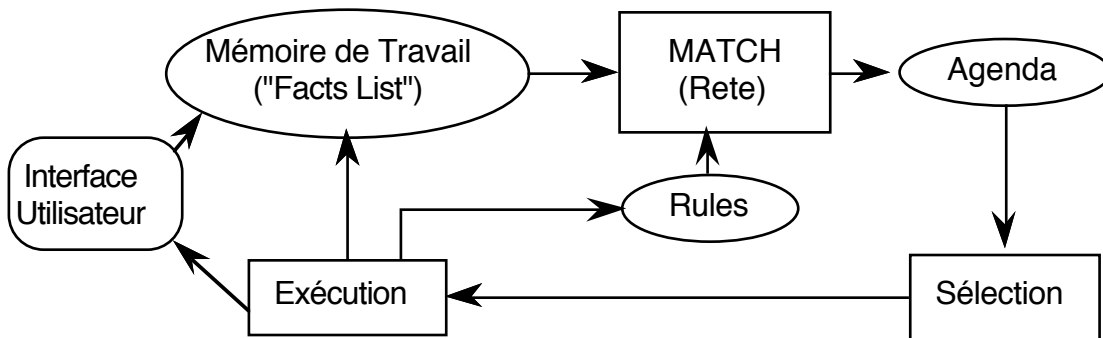
Lesson 6

26 February 2014

## **Rule based programming - Introduction to CLIPS 6.0**

Production System Architecture.....	2
CLIPS : “C Language Integrated Production System” .....	3
defacts.....	3
Rules in CLIPS .....	4
Variables .....	5
Rule Syntax: Constraints .....	9
Predicates .....	11

## 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 part of the rule.

## CLIPS : “C Language Integrated Production System”

### deffacts

A predefined list of facts can be created by the expression "deffacts".

```
(deffacts <NOM> ["<comment>"]
  [(<<FAIT-1>>) (<<FAIT-2>>) ... (<<FAIT-N>>)])
```

the command (reset) will empty the facts list and create an "initial-fact" and to create the list of default facts.

examples :

```
(deftemplate place
  (slot name (type SYMBOL) (default NIL))
  (slot x (type NUMBER) (default -1))
  (slot y (type NUMBER)(default -1))
  (multislot neighbors (default NIL))
)
```

```
(deffacts network-of-places
  (place (name A) (x 0) (y 0) (neighbors B C))
  (place (name B) (x 0) (y 1) (neighbors A D))
  (place (name C) (x 1) (y 0) (neighbors A I))
)
```

```
CLIPS> (deffacts network-of-places
  (place (name A) (x 0) (y 0) (neighbors B C))
  (place (name B) (x 0) (y 1) (neighbors A D))
  (place (name C) (x 1) (y 0) (neighbors A I))
)
CLIPS> (reset)
CLIPS> (facts)
f-0      (initial-fact)
f-1      (place (name A) (x 0) (y 0) (neighbors B C))
f-2      (place (name B) (x 0) (y 1) (neighbors A D))
f-3      (place (name C) (x 1) (y 0) (neighbors A I))
For a total of 4 facts.
```

**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 (RHS)
```

If the rule with the same name exists, it is replaced.  
else the rule is created.

There is no limit to the number of conditions or actions (\* means 0 or more).  
Actions are executed sequentially.

Rules with no condition are activated by (Initial-Fact)

The syntax for condition elements is complex:

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

A condition element (CE) can be a list or a template or user defined object.

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

Deftemplate:

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

A CE can contain constant values or variables.

## Variables

Variables are represented by ?x

There are two sorts of variables in CLIPS:

Index Variables: are assigned the index of a fact that matches a CE.

Attribute Variables: Contain the value of a item that matched a CE.

### Index Variables

Variable : ?x

Index variables are used to identify a fact that has matched a CE

This can be used to retract 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 variables are assigned the value of an item that matched a CE.

These can be used to

- 1) Recover the value for computation
- 2) Detect matching facts.

Syntax for attribute variables.

?var - Defines a variable named var.

The matching value is assigned to ?var.

?\$list - Defines a list of variables named list

? - An unnamed variable. No data is stored.

?\$ - An unnamed list. no data is stored.

WITHIN condition elements, values implicitly bound to variables.

Examples :

```
(assert (a b c))
```

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

```
(assert (d e f))
```

```
(defrule choose-1
```

```
  (a)
```

```
=>
```

```
(printout t "a b c" crlf)
```

```
)
```

```
(defrule choose-1
```

```
  (a b c)
```

```
=>
```

```
(printout t "a b c" crlf)
```

```
)
```

```
(defrule choose-3
```

```
  (a b ?x)
```

```
=>
```

```
(printout t "a b and ?x = " ?x crlf)
```

```
)
```

```
(defrule choose-1-of-3
  (a ?x ?)
=>
(printout t "x = " ?x crlf)
)
```

```
(defrule process-a-list
  (a $?x)
=>
(printout t "The list is " $?x crlf)
)
```

```
(defrule make-a-big-list
  (a $?x $?)
=>
(printout t "The list is " $?x crlf)
)
```

The following "trick" is used to obtain the elements of a list:

```
(defrule process-a-list2
  (a $? ?x $?)
=>
(printout t "x = " ?x crlf)
)
```

```
(defrule increment-x
  ?f <- (a ?x)
=>
  (printout t "x = " ?x crlf)
  (bind ?x (+ ?x 1))
  (printout t "now x = " ?x crlf)
  (retract ?f)
  (assert (a ?x))
)
```

```
(defrule increment-x-example
  ?f <- (a ?x)
=>
  (printout t "x = " ?x crlf)
```

```
(bind ?x (+ ?x 1))
(printout t "now x = " ?x crlf)
(modify ?f (a ?x))
)
```

```
(deftemplate a (slot x))
```

```
(defrule increment-x-exampls
```

```
  ?f <- (a (x ?x))
```

```
=>
```

```
(printout t "x = " ?x crlf)
```

```
(bind ?x (+ ?x 1))
```

```
(printout t "now x = " ?x crlf)
```

```
(modify ?f (x ?x))
```

```
)
```

WITHIN the action part of a rule, values may be assigned by

```
(bind ?Var Value)
```

e.g. (bind ?x 3) assigns 3 to ?x

ATTN: DO NOT use (bind) in condition elements

```
(defrule test
```

```
  ?c <- (a b c)
```

```
=>
```

```
(bind ?c oops)
```

```
(printout t ?c crlf)
```

```
)
```

Rule Activations (associations of a rule with facts that match conditions) are placed on the agenda.

```
(deftemplate person
  "A record for a person"
  (slot family-name)
  (slot first-name)
)
```

```
(assert (person (family-name DOE) (first-name John)))
(assert (person (family-name DOE) (first-name Jane)))
```



```
(defrule Find-same-name
  ?P1 <- (person (family-name ?f) (first-name ?n1))
  ?P2 <- (person (family-name ?f) (first-name ?n2))
=>
  (printout t ?n1 " " ?f" and " ?n2 " " ?f " have the
same family name" crlf)
)
```

```
CLIPS> (assert (person (family-name DOE) (first-name John)))
```

```
<Fact-1>
```

```
CLIPS> (assert (person (family-name DOE) (first-name Jane)))
```

```
<Fact-2>
```

```
CLIPS> (defrule Find-same-name
  ?P1 <- (person (family-name ?f) (first-name ?n1))
  ?P2 <- (person (family-name ?f) (first-name ?n2))
=>
  (printout t ?n1 " " ?f" and " ?n2 " " ?f " have the
same family name" crlf)
)
```

```
CLIPS> (run)
```

```
Jane DOE and Jane DOE have the same family name
```

```
Jane DOE and John DOE have the same family name
```

```
John DOE and Jane DOE have the same family name
```

```
John DOE and John DOE have the same family name
```

Question: Why does the rule execute 4 times?

## 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

```
(defrule Find-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)
)
```

example :

(?x & green | blue) - ?x must be green or blue for the condition to match

(?x & ~red) - ?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
(?xl: (<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

```
(defrule Find-same-name
  ?P1 <- (person (family-name ?f) (first-name ?n1))
  ?P2 <- (person (family-name ?f)
           (first-name ?n2&:(neq ?n1 ?n2)))

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

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-4  
  (data ?y)  
  not (data ?x&:(> ?x ?y))  
=>)
```

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