

# Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 / MoSIG M1

Second Semester 2016/2017

Lesson 14

29 March 2017

## **Rule based programming - Forward chaining**

Production Systems .....	2
Production System architecture .....	3
CLIPS : “C Language Integrated Production System” .....	4
The Facts List .....	5
Deftemplates.....	6
deffacts .....	8
Rules in CLIPS .....	9
Variables .....	9
Rule Syntax: Constraints .....	14
Predicates .....	15
The ACTION part (RHS) of a rule .....	17
System Actions .....	17

## Production Systems

Three techniques are commonly used to represent knowledge:

- Rules
- Schema systems
- Logic

Many modern AI programming environment combine all three.

Rules are typically expressed as:

if <CONDITION> then <ACTION>

Rules are often used to program procedural or reactive intelligence.

Rule based programming languages take their inspiration from the observation of conditioned reflexes in animals and humans.

Conditioned Reflexes are trained associations (PAVLOV)

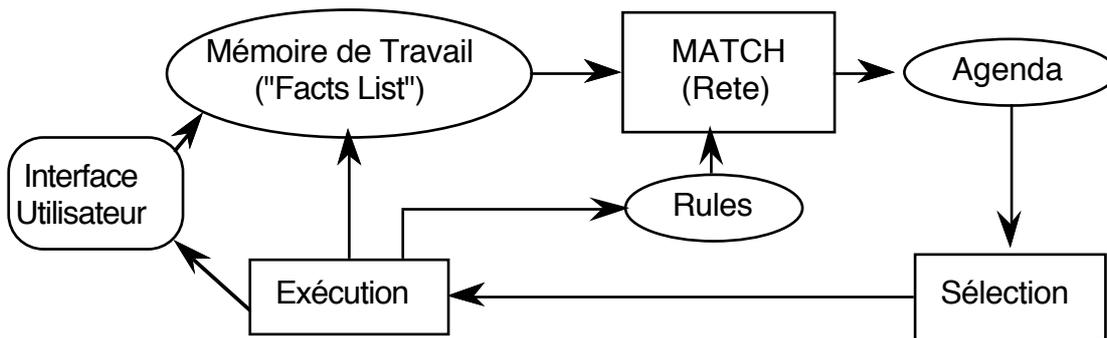
Condition  $\Rightarrow$  Reaction

A reaction can be the assertion of a fact or a physical action.

The association of facts and rules poses an important problem for rule based programming environments. If the rules can contain variables, then the algorithmic complexity is exponential in the number of conditions.

In this lesson we will learn of a "fast" algorithm for matching rules to Facts: RETE.

In 1975, Alan Newell at CMU took on a doctoral student named Laney Forgy. Newell asked Forgy to create a programming environment for rule based programming. The result was a system named "OPS" (Operational Production System")

**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.

Newell and Forgy soon discovered that in a such a system, adding variables to the rules greatly enhances the expressive power, but at the same time causes a problem of algorithmic complexity in the matching of rules to facts. Newell asked Forgy to develop a fast algorithm for matching rules to Facts. The result is the RETE algorithm.

In this architecture, each fact is labeled with an index. In each cycle, the **CONDITION** part of rules are matched to the facts. The association of facts and rules are stored in an agenda. One of the associations is then selected for execution.

The list of associations of facts and rules are stored in an agenda. One of the associations is selected for execution.

There are several different models for how to sort the agenda. The most popular is to use a stack "Last In First Out" or LIFO.

Rule based programming languages:

OPS-5 ('78) => ART ('80) => CLIPS ('85) => CLIPS 6 ('94)

The OPS system was used to construct a very successful expert system for configuring VAX computers (R1). R1 gained several hundred million dollars for Digital Equipment Corporation from 1980 to 1985.

**CLIPS : “C Language Integrated Production System”**

The OPS system was commercialized by DEC, but also by Teknowledge under the name "ART" Automated Reasoning Tool. NASA selected ART as its standard. Unfortunately ART was built in the LISP language and was very expensive.

(50 K\$ for the license, 50 K\$ for the LISP machine).

In 1985, NASA engineers reprogrammed ART in C. The result was called CLIPS.

The origin of CLIPS is a C language version of the ART system (Automated Reasoning Tool) programmed by NASA engineers.

OPS, ART and CLIPS all share the same architecture, and use the RETE algorithm.

As a public organization, NASA distributed CLIPS for free (Before GNU and ShareWare)

Successive versions of CLIPs have been distributed and a substantial user community has emerged. We will use a version, CLIPS 6.0, that is programmed using object oriented programming. This version integrates many different knowledge representation styles, including Rules, Schema, and a logic based Truth Maintenance system.

For NASA, CLIPS is used to construct Expert Systems for

- 1) Process Control
- 2) On-Board Error Diagnostic
- 3) Mission Planning
- 4) Logistics Planning.

## The Facts List

The working memory of CLIPS is called the "Facts List".

A Fact can represent:

- A list of items (PRIMITIVES including symbols, strings, integers or floats)
- A template composed of a "name" and a list of "attribute-value" pairs.
- An object from the CLIPS object oriented system.

CLIPS' working memory is represented as a list of facts.

Facts are a list or structure or items (or an object).

Items belong to a hierarchy of 13 primitive types.

STRING

It is also possible to define new types. Version 6.0 of CLIPS contains an object-oriented environment named COOL. ("C-Language Object Oriented Language").

User defined types belong to the class COOL class USER

In later lessons we will use the COOL. For now, we will work with lists and structures.

Facts, in CLIPS, are created and destroyed by the commands:

ASSERT: (ASSERT <<FACT>>)

RETRACT, (RETRACT <<INDEX>>)

(RESET) - Empties the working memory

(CLEAR) - Resets the entire system to empty.

Typing (RESET) creates an "initial fact" with index 0.

Typing (Clear) deletes all templates, rules, facts, classes etc..  
And then creates an "initial fact" with index 0.

f-0 (initial-fact)

There are 5 keywords that cannot be used in the first field in a fact:  
Test, and, or, not, declare.

Each fact is identified by a unique "index". Indices are allocated in order. They are fundamental to the logic of CLIPS.

(assert (a b c))

ATTN (assert (1 2 3)) will generate an error. The first item must be a symbol

## Deftemplates

A template is a set of "attribute - value" pairs. The attribute name are used as an index into the template. Such structures are defined with deftemplate.

A template makes it possible to manipulate attributes of a fact without having to scan through a list. Attributes are called "slots" in clips.

Example:

```
(deftemplate person          ; A template for a person
  "template for a person" ; Optional Comment
  (slot name                 ; Name of a person
    (default "John Doe"))   ; Default name
  (slot age                  ; age of the person
    (type INTEGER))        ; default will be 0
)
```

Instances of templates are created by (assert).

```
(assert (person))
(assert (person (name "joe")))
(assert (person (name "jim") (age 39)))
```

A number of commands exist inspecting templates

```
(list-deftemplates)
```

```
(ppdeftemplate <template-name>)
(undeftemplate <template-name>)
```

examples:

```
CLIPS> (list-deftemplates)
initial-fact
person
For a total of 2 deftemplates.
CLIPS> (ppdeftemplate person)
(deftemplate MAIN::person "template for a person"
  (slot name (default "John Doe"))
  (slot age (type INTEGER)))
CLIPS>
```

We can define types and default values for slots.

We can also define a list of allowed symbols for a slot value.

example :

```
(deftemplate person          ; A template for a person
"template for a person" ; Optional Comment
  (slot name                ; Names of a person
  (type STRING)
  (default "Pierre Dupont")) ; Default name
  (slot age                  ; age of the person
  (type INTEGER)            ; default will be 0
  (range 0 120))           ; Possible ages
  (slot Profession          ;
  (type SYMBOL)
  (allowed-values artist engineer salesman manager
  other none)
  (default none)
)
)
```

For numbers we can define a range

```
(slot age
  (type NUMBER)
  (range 0 120))
```

**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)
)
```

### 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-example
  ?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?

Answer: Twice because a fact can match itself and twice because the rule matches Jane with John as well as John with Jane

```

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

```

```

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

```

### 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))
  =>)
```

**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.

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 ?firstname (read))
  (printout t "Family name? ")
  (assert (person ?firstname (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")
```

)

4) Halt :        Stop execution.