Intelligent Systems: Reasoning and Recognition

James L. Crowley

Ensimag 2 Second Semester 2018/2019

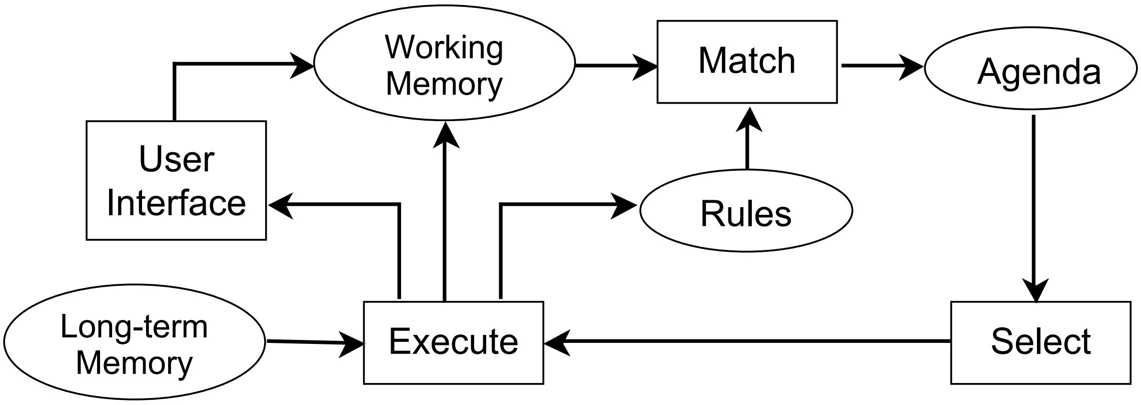
Lesson 16 5 April 2019

Knowledge Representation and Reasoning With Rule-Based Production Systems

Production System Architecture (Reminder)

In a production system, data and concepts in working memory are associated with data and procedures in long-term memory.

Rules are encoded as condition-action pairs: Condition* ⇒ Action*



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 working memory to conditions of rules to produce "activations" (associations of WM and rules).
- Select: An activation is selected for execution.
- Execute: Execute the actions specified in the activation.

The list of activations (associations of facts and rules) is 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.

Rules in CLIPS

CLIPS rules allow programming of reactive in the form of Rules.
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 with the new rule.

Each line of the condition is referred to as a "Condition element (CE)
A condition element can match a list or a template or a user-defined object.
CE can match 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 an item that matches a CE.

Index Variables

Index variables are used to identify a fact that has matched a CE. The variable is assigned the index of the fact. This index be used to retract of modify the fact.

Attribute Variables

Attribute variables are assigned the value of an data field that matched a variable in a CE. Attribute variables can be used to

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

Attribute variables can have 3 forms:

?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 are implicitly bound to variables.
IT is NOT necessary to assign the value to the variable. The assignment is implicit.

Examples :

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

(defrule a
   (a)  ;; matches only facts with a single item. (a)
=>
(printout t "a" found crlf)
)

(defrule abc
   (a b c)  ;; matches only facts of the form (a b c)
=>
(printout t "a b c" crlf)
)

(defrule abx
   (a b ?x)
=>
(printout t "a b  and ?x = " ?x crlf)
)

(defrule ax-
   (a ?x  ?)
=>
(printout t "x = " ?x crlf)
)
```

Scanning a list

The following "trick" is used to obtain each item from a list:

```
(defrule Extract-every-element-from-list
   (a $? ?x $?)
=>
(printout t "x = " ?x crlf)
)
$?   Matches nothing, or 1 item, or 2 items etc...
?x   matches the next item.
$?    Matches the rest of the list.
```

This rule will provide one activation for each item after the first item in the fact.

```
(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))
)

(clear)

(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
A
ctivations (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 (neq ?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
`(?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

```
(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.

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)))
)
```

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.

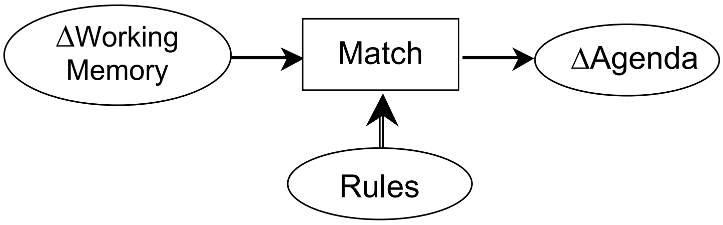
example :

```
(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)))
)
```

The RETE Matching Algorithm

In a production system, in principle, each condition element of each rule requires a complete scan of the working memory during each cycle of execution. This can be very costly. The RETE algorithm avoids this by providing incremental matching between facts and conditions of rule.



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.

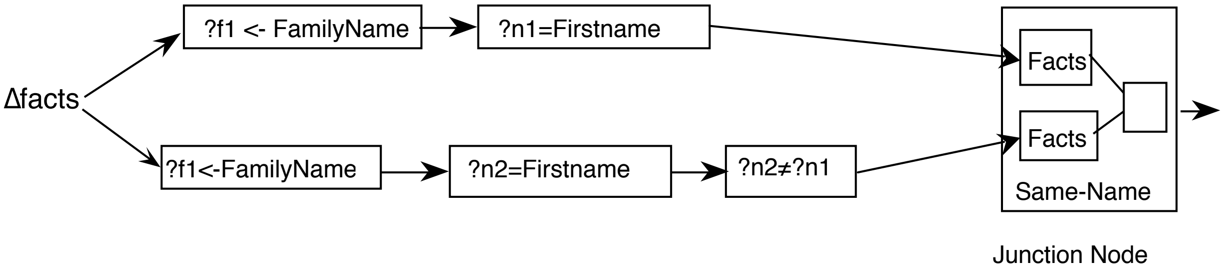
Changes in working memory filter through this decision network generate changes to the agenda.

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

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)
```



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

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, using even simple embedded computing hardware.

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.

```
(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