Block 1 – Knowledge Representation and Search

Chapter 2:
Inference and control in RBS.
RETE.

Block1: Chapter 2- Index

1. Inference engine
2. Conflict resolution strategies
3. Examples
4. *Rete* match algorithm
5. Exercises.
6. Annex: BNF syntax of CLIPS rules

Bibliography

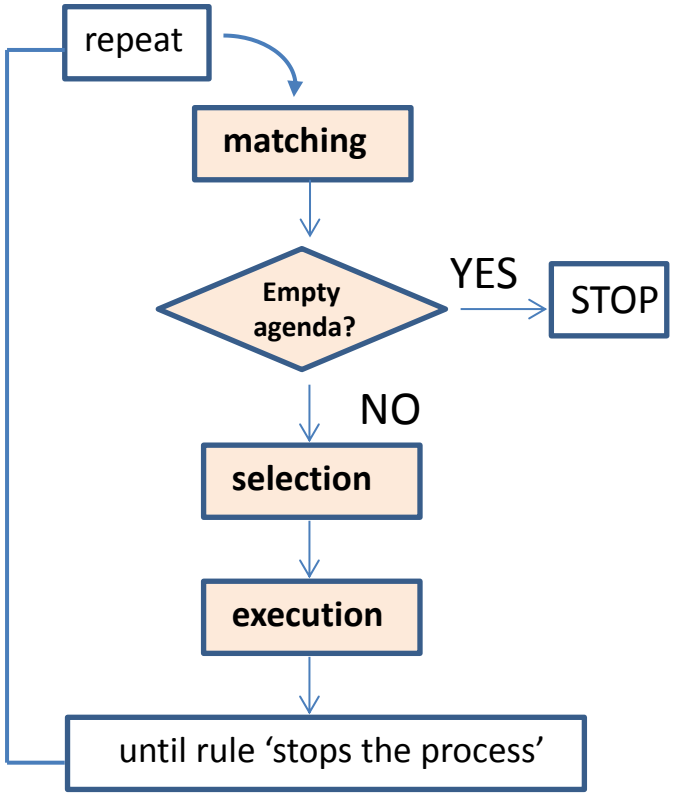
- Capítulo 3: *Sistemas Basados en Reglas*. Inteligencia Artificial. Técnicas, métodos y aplicaciones. McGraw Hill, 2008.
- Charles Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", *Artificial Intelligence*, 19, pp 17–37, 1982.
- CLIPS User's guide
- CLIPS Basic Programming guide

1. Inference engine

Unlike programming logic, which follows a Declarative semantics, RBS follow a Procedural semantics. This difference is due to the procedural nature of the RHS of the rules.

CLIPS uses a forward chaining inference engine based on the Rete match algorithm (Rete-based inference engine)

The control mechanism is referred to as the ***recognize-act cycle*** of a Rete-based inference engine:



Rule matching: generates *a Conflict Set* or *Agenda* with all the applicable rules or rule instances found.

If the Agenda is empty the process stops. Otherwise, a rule instance is selected from the *Conflict Set* according to a predetermined criterion (Conflict Resolution Strategy)

The RHS of the selected instance is executed, updating the Working Memory in accordance or executing the external actions.

1. Inference engine

WM = Working Memory; RB= Rule Base; CSet= Conflict Set; RuleInst=rule instance

```
WM=initial facts
CSet=Matching(WM,RB)
while goal ⊄ WM and CSet ≠ ∅
    RuleInst=SelectOneRuleInstance(CSet)
    WM=WM \ retracts of RuleInst    ;; Execution of the RHS (1)
    WM=WM ∪ asserts of RuleInst     ;; Execution of the RHS (2)
    CSet=CSet \ rule instances eliminated by the retracts (3)
    CSet=CSet ∪ Matching(WM,RB)                          (4)
end_while
if goal ⊆ WM        ;; a rule fires which specifically tells the system to halt
    then SUCCESS
    else FAILURE
```

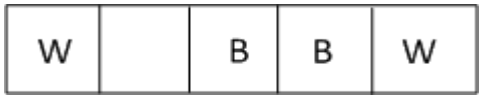
(1) (2): execution of the RHS of RuleInst; execute the 'retract' and 'assert' commands and update the WM accordingly.

(3) Update the CSet by eliminating the rule instances that depend on the eliminated facts (retract)

(4) Rule-matching phase with the newly generated facts (assert)

1. Inference engine (example 1)

Linear puzzle:



Initial situation

Final situation

Rule Base

```
(defrule emtpy-1-left
    ?f1 <-  (puzzle $?x ?y E $?z)
  =>
    (retract ?f1)
    (assert (puzzle $?x E ?y $?z)))

(defrule emtpy-2-left
    ?f1 <- (puzzle $?x ?y1 ?y2 E $?z)
  =>
    (retract ?f1)
    (assert (puzzle $?x E ?y2 ?y1 $?z)))

(defrule emtpy-1-right
    ?f1 <- (puzzle $?x E ?y $?z)
  =>
    (retract ?f1)
    (assert (puzzle $?x ?y E $?z)))

(defrule emtpy-2-right
    ?f1 <- (puzzle $?x E ?y1 ?y2 $?z)
  =>
    (retract ?f1)
    (assert (puzzle $?x ?y2 ?y1 E $?z)))
```

Working memory

```
(deffacts data
    (puzzle W E B B W))
```

→ f-1: (puzzle W E B B W)

Matching

empty-1-left: f-1, ?y=W, $?x=(), $?z=(B B W), ?f1=1
empty-1-right: f-1, ?y=B, $?x=(W), $?z=(B W), ?f1=1
empty-2-right: f-1, ?y1=B, ?y2=B, $?x=(W), $?z=(W), ?f1=1

Agenda or Conflict Set (rule instances)

Selection

empty-1-left: f-1, ?y=W, $?x=(), $?z=(B B W), ?f1=1
empty-1-right: f-1, ?y=B, $?x=(W), $?z=(B W), ?f1=1
empty-2-right: f-1, ?y1=B, ?y2=B, $?x=(W), $?z=(W), ?f1=1

1. Inference engine (example 1)

Execution

~~empty-1-right: f-1, ?y=B, $?x=(W), $?z=(B W), ?f1=1~~
~~empty-2-right: f-1, ?y1=B, ?y2=B, $?x=(W), $?z=(W), ?f1=1~~

Working memory

f-2: (puzzle E W B B W)

Matching

empty-1-right: f-2, ?y=W, $?x=(), $?z=(B B W), ?f1= 2
empty-2-right: f-2, ?y1=W, ?y2=B, $?x=(), $?Z=(B W), ?f1= 2

Selection

empty-1-right: f-2, ?y=W, $?x=(), $?z=(B B W), ?f1= 2
empty-2-right: f-2, ?y1=W, ?y2=B, $?x=(), $?Z=(B W), ?f1= 2

Execution

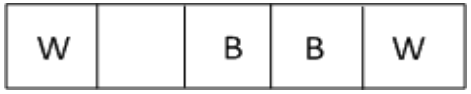
~~empty-2-right: f-2, ?y1=W, ?y2=B, $?x=(), $?Z=(B W), ?f1= 2~~

Working memory

f-3: (puzzle W E B B W)

1. Inference engine (example 2)

Linear puzzle:



Initial situation

Final situation

Working memory

```
(deffacts data
    (cell 1 W)
    (cell 2 E)
    (cell 3 B)
    (cell 4 B)
    (cell 5 W))
```

→

```
f-1: (cell 1 W)
f-2: (cell 2 E)
f-3: (cell 3 B)
f-4: (cell 4 B)
f-5: (cell 5 W)
```

Rule Base

```
(defrule emtpy-1-left
    ?f1 <-  (cell ?x E)
    ?f2 <- (cell ?y ?cell)
    (test (= ?y (- ?x 1)))
  =>
    (retract ?f1 ?f2)
    (assert (cell ?y E))
    (assert (cell ?x ?cell)))
```

```
(defrule emtpy-2-left
    ?f1 <-  (cell ?x E)
    ?f2 <- (cell ?y ?cell)
    (test (= ?y (- ?x 2)))
  =>
    (retract ?f1 ?f2)
    (assert (cell ?y E))
    (assert (cell ?x ?cell)))
```

. . . . .

1. Inference engine (example 2)

Matching

empty-1-left: f-2, f-1, ?x=2, ?y=1, ?f1=2, ?f2=1
empty-1-right: f-2, f-3, , ?x=2, ?y=3, ?f1=2, ?f2=3
empty-2-right: f-2, f-4 , ?x=2, ?y=4, ?f1=2, ?f2=4

Selection

empty-1-left: f-2, f-1, ?x=2, ?y=1, ?f1=2, ?f2=1
empty-1-right: f-2, f-3, , ?x=2, ?y=3, ?f1=2, ?f2=3
empty-2-right: f-2, f-4 , ?x=2, ?y=4, ?f1=2, ?f2=4

Execution

~~empty-1-right: f-2, f-3, , ?x=2, ?y=3, ?f1=2, ?f2=3~~
~~empty-2-right: f-2, f-4 , ?x=2, ?y=4, ?f1=2, ?f2=4~~

Working memory

f-3: (cell 3 B)
f-4: (cell 4 B)
f-5: (cell 5 W)
f-6: (cell 1 E)
f-7: (cell 2 W)

Matching

empty-1-right: : f-6, f-7, ?x=1, ?y=2, ?f1=6, ?f2=7
empty-2-right: f-6, f-3, ?x=1, ?y=3, ?f1=6, ?f2=3

2. Conflict resolution strategies

The Conflict Set or Agenda is a collection of rule instances or activations which are those rules which match pattern entities. Zero o more rule instances can be on the Agenda. The top activation on the agenda is always selected for execution.

Conflict resolution strategy: criterion to select the activation from the Agenda

Refraction

A rule can only be used once with the same set of facts (same fact indexes) in the WM and same variable bindings. Whenever WM is modified, all rules can again be used provided that, **at least**, a new fact is generated that causes a new activation of the rule. This strategy prevents a single rule and list of facts from being used repeatedly, resulting in an infinite loop of reasoning.

CLIPS applies refraction and does not allow a rule to be fired twice on the same data (same fact indexes and same variable bindings) thus avoiding firing rules over and over again on exactly the same facts.

This behaviour is also supported by the default option of CLIPS of not allowing fact duplication. CLIPS does not accept a duplicate fact, i.e., a fact with the same structure but a different index that other fact that is already in the WM.

2. Conflict resolution strategies: example refraction and duplicated facts

```
(defrule R1
      (lista $?x ?y ?z $?w)
      (test (evenp ?z))
   =>
      (assert (lista $?x ?z ?y $?w)))
```

WM= { f-1: (lista 12 5 24 7)}

Matching

~~R1: f-1, $?x=(12), ?y=5, ?z=24, $?w=(7)~~

Selection **Execution**

WM= { f-1: (lista 12 5 24 7)
f-2: (lista 12 24 5 7)}

Matching

~~R1: f-2, $?x=(), ?y=12, ?z=24, $?w=(5 7)~~

Selection **Execution**

WM= { f-1: (lista 12 5 24 7)
f-2: (lista 12 24 5 7)
f-3: (lista 24 12 5 7)}

The activation R1: f-1, $?x=(12), ?y=5, ?z=24, $?w=(7) is not inserted again.

Matching

~~R1: f-3, $?x=(), ?y=24, ?z=12, $?w=(5 7)~~

Selection **Execution**

WM= { f-1: (lista 12 5 24 7)
f-2: (lista 12 24 5 7)
f-3: (lista 24 12 5 7)}

CLIPS does not insert again the fact (lista 12 24 5 7)

2. Conflict resolution strategies: agenda

If the agenda contains more than one rule instance, the conflict resolution strategy decides which one to select. **CLIPS always selects the top activation of the Agenda to be executed**.

The conflict resolution strategy determines how rule instances are ordered in the Agenda.

Explicit priority

Numeric value: salience of rules

In CLIPS, possible salience values range from -10,000 to 10,000. The higher the value, the higher the priority of the rule. If no salience is specified in the rules, all rules have salience 0.

```
(defrule <rule name> ["comment"]
    (declare (salience <integer>))
        <conditional-element>*      ; left-hand side (LHS)  of the rule
                                     ; conditions to satify
=>

....
```

2. Conflict resolution strategies

Example: we want to gather 5 people to play basket, preferably tall people over 2m; otherwise, pick people below 2m.

```
(defrule over-2m
    (declare (salience 30))
?f1 <-  (height ?per ?tall)
        (test (>= ?tall 2))
?f2 <- (count ?numper)
 =>
    (retract ?f1 ?f2)
    (assert (count (+ ?numper 1))))

(defrule below-2m
     (declare (salience 10))
   ?f1<-  (height ?per ?tall)
   ?f2 <-  (count ?numper)
 =>
    (retract ?f1 ?f2)
    (assert (count (+ ?numper 1))))
```

```
(defrule final-1
     (declare (salience 100))
        (count 5)
=>
  (halt)
  (printout t "We already have 5 people to play basket " crlf)

(defrule final-2
  =>
   (halt)
   (printout t "We were not able to find 5 people to play basket ", crlf))

(deffacts data
    (height John 2.02) (height Peter 1.92)(height Terry 1.86)
    (height Lucas 2.01)(height Nick 2.05)(height Joshua 1.94)
    (count 0))
```

2. Conflict resolution strategies

Depth strategy

Newly activated rules are managed through a priority queue. Ties are broken by using a LIFO strategy; i.e., more priority to most recent facts and rule instances.

Breadth strategy

Newly activated rules are managed through a priority queue. Ties are broken by using a FIFO strategy; i.e., more priority to older facts and rule instances.

Random

Each activation is assigned a random number which is used to determine its placement among activations of equal salience. CLIPS also allows to select Random as conflict resolution strategy.

Specificity

Use the most specific rule, the one with most detail or constraints. The specificity of a rule is determined by the number of comparisons that must be performed on the LHS of the rule. Each comparison to a constant or previously bound variable adds one to the specificity. Each function call made on the LHS of a rule as part of a test conditional element adds one to the specificity.

CLIPS allows to select **Simplicity** and **Complexity** as conflict resolution strategies.

Simplicity: Ties are broken by giving more priority to rules with equal or lower specificity.
Complexity: Ties are broken by giving more priority to rules with equal or higher specificity

3. Examples

The next slides show 4 different examples of the application of the recognize-act cycle of a Rete-based inference engine.

The goal is to trace the behaviour of the a Rete-based inference engine when put to work in a specific RBS.

For each example, we define a) the initial WM, b) the rules that make up the RB and c) the conflict resolution strategy to use.

The objective is to see the successive application of the *match-selection-execution* cycle until the problem is finished (in these examples, the problem is over when the agenda is empty).

3. Example 1

Assuming the Agenda works in **breadth-first**, trace the following RBS and say the final contents of the WM. This RBS orders a list composed of several non-ordered numbers.

WM={(list 3 2 7 5)}

```
(defrule R1
    ?f <- (list $?x ?y ?z $?w)
    (test (< ?z ?y))
=>
    (assert (list $?x ?z ?y $?w)))
```

WM (facts)	Agenda (rule instances)	
f-1: (list 3 2 7 5)	R1: f-1 {$?x=(3 2),?y=7, ?z=5, $?w=(), ?f=1}	selection (1)
	R1: f-1 {$?x=(),?y=3, ?z=2, $?w=(7 5), ?f=1}	selection (2)
f-2: (list 3 2 5 7)	R1: f-2 {$?x=(),?y=3, ?z=2, $?w=(5 7), ?f=2}	selection (3)
f-3: (list 2 3 7 5)	R1: f-3 {$?x=(2 3),?y=7, ?z=5, $?w=(), ?f=3}	selection (4)
f-4: (list 2 3 5 7)	no match	
no more facts (duplicated fact)		

WM final={(list 3 2 7 5) (list 3 2 5 7) (list 2 3 7 5)(list 2 3 5 7)}

3. Example 2

Assuming the Agenda works in **breadth-first**, trace the following RBS and say the final contents of the WM. This RBS orders a list composed of several non-ordered numbers.

WM={(list 3 2 7 5)}

```
(defrule R1
    ?f <- (list $?x ?y ?z $?w)
    (test (< ?z ?y))
=>
    (retract ?f)
    (assert (list $?x ?z ?y $?w)))
```

	WM (facts)	Agenda (rule instances)	
eliminate (1) →	f-1: (list 3 2 7 5)	R1: f-1 {$?x=(3 2),?y=7, ?z=5, $?w=(), ?f=1}	← selection (1)
		R1: f-1 {$?x=(),?y=3, ?z=2, $?w=(7 5), ?f=1}	← eliminate (1)
eliminate (2) →	f-2: (list 3 2 5 7)	R1: f-2 {$?x=(),?y=3, ?z=2, $?w=(5 7), ?f=2}	← selection (2)
	f-3: (list 2 3 5 7)	no match	

WM final={(list 2 3 5 7)}

3. Example 3

Assuming the Agenda works in **breadth-first** (more priority to older rule instances and facts starting by rule 'move'), trace the following RBS and say the final contents of the WM.

WM={(list a b c d e) (move-to-front c)}

```
(defrule move
   ?m<- (move-to-front ?who)
   ?l <- (list $?front ?who $?rear)
=>
   (assert (list ?who $?front $?rear))
   (assert (change-list yes)))
```

```
(defrule print
    ?ch <- (change-list yes)
    (list $?list)
=>
    (retract ?ch)
    (printout t "List is " $?list crlf))
```

WM (facts)	Agenda (rule instances)
f-1: (list a b c d e) f-2: (move-to-front c)	move: f-2,f-1 {?who=c, $?front=(a b), $?rear=(d e) ?m=2, ?l=1}
f-3: (list c a b d e) f-4: (change-list yes)	move: f-2,f-3 {?who=c, $?front=(), $?rear=(a b d e) ?m=2, ?l=3} print: f-4,f-1 {$?list=(a b c d e), ?ch=4} print: f-4,f-3 {$?list=(c a b d e), ?ch=4}
(duplicated fact)	

selection (1)

selection (2)

selection (3)

eliminate (3)

eliminate (3)

```
CLIPS> (run)
List is (a b c d e)
```

WM final={(list a b c d e)(move-to-front c)(list c a b d e)}

3. Example 4

Assuming the Agenda works in **depth-first** (more priority to newer rule instances and facts starting by rule 'move'), , trace the following RBS and say the final contents of the WM.

WM={(list a b c d e) (move-to-front c)}

```
(defrule move
   ?m<- (move-to-front ?who)
   ?l <- (list $?front ?who $?rear)
=>
   (assert (list ?who $?front $?rear))
   (assert (change-list yes)))
```

```
(defrule print
    ?ch <- (change-list yes)
     (list $?list)
=>
     (retract ?ch)
     (printout t "List is " $?list crlf))
```

WM (facts)	Agenda (rule instances)
f-1: (list a b c d e) f-2: (move-to-front c)	move: f-2,f-1 {?who=c, $?front=(a b), $?rear=(d e) ?m=2, ?l=1}
f-3: (list c a b d e) f-4: (change-list yes)	print: f-4,f-3 {$?list=(c a b d e), ?ch=4} print: f-4,f-1 {$?list=(a b c d e), ?ch=4} move: f-2,f-3 {?who=c, $?front=(), $?rear=(a b d e) ?m=2, ?l=3}
(duplicated fact)	

selection (1)

selection (2)

eliminate (2)

eliminate (2)

selection (3)

```
CLIPS> (run)
List is (c a b d e)
```

3. Example 4 (continuation)

WM (facts)	Agenda (rule instances)
f-1: (list a b c d e) f-2: (move-to-front c)	
f-3: (list c a b d e)	
f-5: (change-list yes)	print: f-5,f-3 {$?list=(c a b d e), ?ch=5} ← selection (4) print: f-5,f-1 {$?list=(a b c d e), ?ch=5} ← eliminate (4)

eliminate (4) →

CLIPS> (run)
List is (c a b d e)

3. Examples (conclusions)

Choice of conflict resolution strategy, specifically when there are retractions of facts, can make a difference in the system performance.

Because of the effect of conflict resolution strategies, rules interact and the order of rule firing matters.

One must go beyond the declarative meaning of the rules and consider when (under which circumstances) they will fire.

One cannot properly understand a rule simply by reading it in isolation; one must consider the related rules and the conflict resolution strategy as well.

4. *Rete* match algorithm

Rete match algorithm was devised by Charles Forgy of Carnegie-Mellon University. The term *Rete* is Latin for 'net', and describes the software architecture of the pattern-matching process [Forgy 82].

Rete represents the rules internally as data in the so-called *Rete network*. The compiler creates the *Rete* network from the rules specified in the RBS.

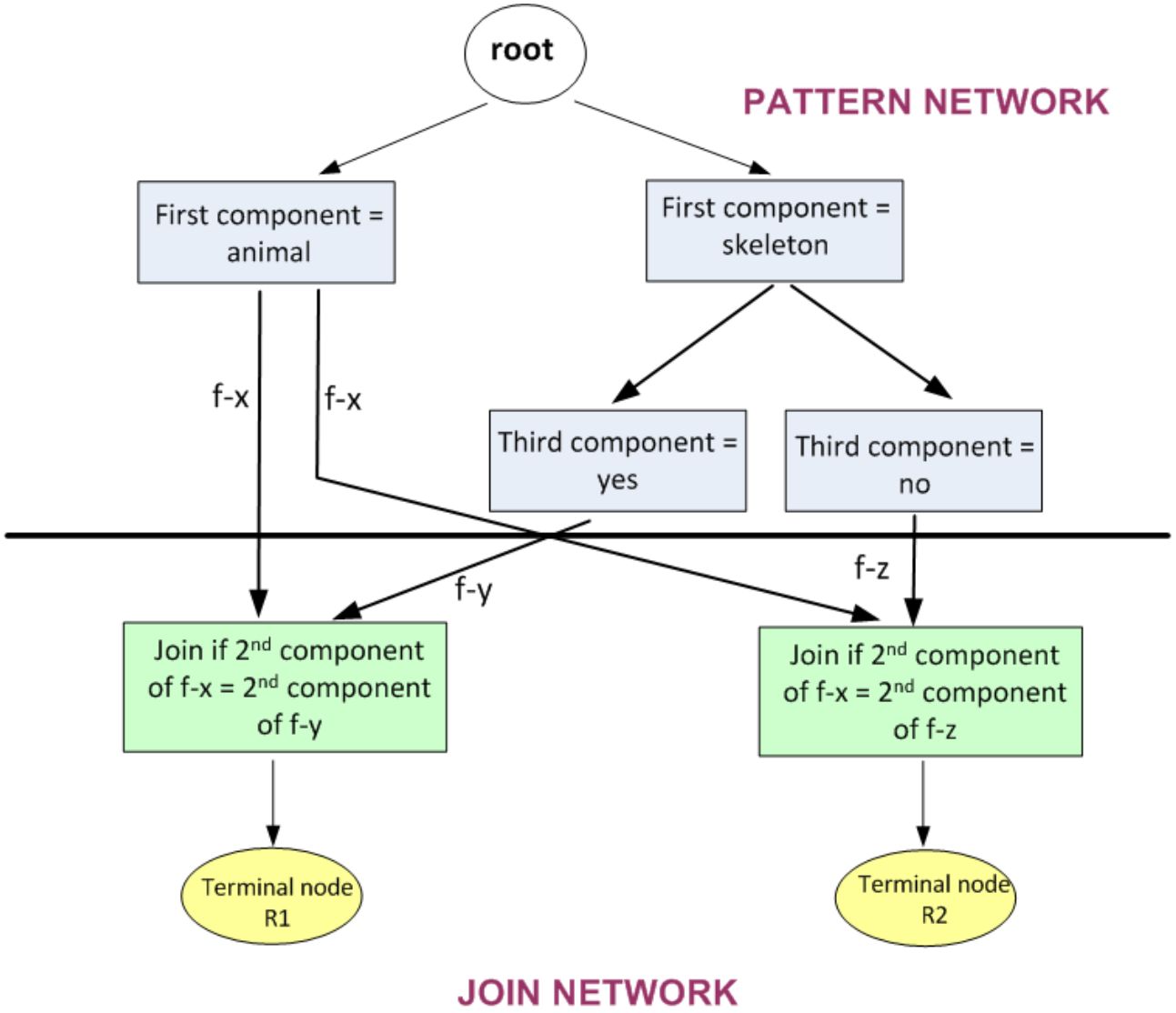
The *Rete* network consists of a *pattern network* and a *join network*.

The *Rete* network amounts to a finite state machine that consumes modifications of facts.

4. *Rete* network

```
(defrule R1
      (animal ?a)
      (skeleton ?a yes)
 =>
      (assert (vertebrate ?a)))

(defrule R2
     (animal ?a)
     (skeleton ?a no)
=>
     (assert (invertebrate ?a)))
```



4. *Rete* match algorithm: properties

Rete exploits two properties (common to all RBS):

Structural similarity

- Rule conditions contain many similar patterns although not identical.
- *Rete* takes advantage of *structural similarity* in rules (the fact that many rules often contain similar patterns or groups of patterns) by pooling common components so they need not be computed more than once.
- A compiler groups common patterns before executing a RBS. This compiler generates the *Rete* network.

Temporal redundancy

- Facts in a RBS change slowly over time. Few changes occur in the WM at each cycle even though the WM contains many facts.
- *Rete* takes advantage of *temporal redundancy* by remembering what has already been matched from cycle to cycle and computing only the changes necessary for the newly added or newly removed facts (rules remain static and facts change).
- It saves the state of the matching process during one cycle and re-computes the changes in this state only for the changes that occur in the WM.
- This way, the computational cost of the RBS depends on the frequency of changes in the WM, which is usually low, and not on the size of its contents.

5. Exercises

Exercise 1

Given a list of integer numbers, we want to obtain a new list where those numbers whose values do not coincide with their ordering positions are replaced by 0 (we assume the first number in the list takes up position number 1). For example, given the list (list 15 2 4 6 5 3 7 8 15) the new list would be (list 0 2 0 0 5 0 7 8 0). Write a single rule in CLIPS to perform this task and trace the resulting RBS.

Exercise 2

Given a fact that represents a list of integer and non-ordered numbers, which may contain repeated numbers several times, write a single rule (if possible) to get the same initial list of numbers without repetitions.

Example:

Initial WM: (list 1 3 4 5 6 7 4 6 4 4 3 4 5 6 4 5 6 4 8 5 7 3 2 4 4 4 1 5 6 7 1 2 3 2 3 4 6 5 3 4 5 1 4 3)

Final WM: (list 2 7 6 5 4 3 1 8)

Exercise 3

Given two sequences of DNA, write a RBS (only one rule, if possible) that counts the number of mutations (you can use a global variable to store the number of mutations or you can create a new fact that keeps this value). For example: given the two DNA sequences

(A A C C T C G A A A) and (A G G C T A G A A A)

there are three mutations.

5. Exercises

Exercise 4

Let WMinitial={(lista 1 2 3 4)} be the initial WM of a RBS whose RB is composed of the following two rules:

```
(defrule R1
   ?f <- (lista ?x $?z)
 =>
  (retract ?f)
  (assert (lista $?z))
  (assert (elemento ?x)))
```

```
(defrule R2
     ?f <- (elemento ?x)
           (elemento ?y)
           (test (< ?x ?y))
      =>
           (retract ?f)
           (assert (lista-new ?x ?y))
```

, assuming a breadth-first strategy, i.e., more priority to older facts and rule instances, starting by R1:

a) Which will be the final WM? Show the trace that follows from the inference process.

b) If the initial WM were WM= {(lista 1 2 2 4)}, how many facts (lista-new ....) would be in the final WM? Which ones?

c) Assuming again the initial WMinitial= {(lista 1 2 3 4)}, if we eliminate the command (retract ?f) from R1, which changes regarding the facts (lista-new ...) will be in the final WM ?

d) Assuming again the initial WMinitial= {(lista 1 2 3 4)}, if we eliminate the command (retract ?f) from R2, which changes regarding the facts (lista-new ...) will be in the final WM?

5. Exercises

Exercise 5

Write a simple RBS (only one rule, if possible) to find out how many correct numbers a board of the Loteria Primitiva has (this is a weekly lottery in Spain where players choose six numbers out of a total of 49). The initial WM contains two facts, one fact shows the winning number combination and the other shows the six numbers chosen by the player (you can use a global variable or you can create a new fact to count the number of wins). Here is one example of initial WM:

Initial WM = {(winning-board 2 5 8 13 24 35) (chosen-numbers 3 5 15 24 26 37)}

Show a trace of the resulting RBS.

Let's suppose the player does not introduce numbers in increasing order, would the RBS you've designed above work equally in this case? Why?

Exercise 6

A bingo board consists of 5 rows each containing 5 numbers. The WM of a RBS contains a fact which represents a winner line, for example, (winner-line 12 21 34 56 77). Write a fact
to represent the bingo board and a production rule to determine if there exists a winner line on the bingo board.

6. Annex: BNF syntax of CLIPS rules

```
(defrule <rule-name> [<comment>]
    [<declaration>]
    <conditional-element>*
=>
   <action>*)
```

6. Annex: BNF syntax of CLIPS rules (LHS of rules)

<declaration> ::= (declare (salience <integer-expression>))

<conditional-element>::= (and <CE>) | (or <CE>) | (not <CE>)| (test <function-call>) |
<ordered-pattern> | <assigned-pattern>

<assigned-pattern> ::= <single-vble> ← <ordered-pattern>

<ordered-pattern> ::= (<symbol> <constraint>*)

<constraint> ::= ? | $? | <constant> | <variable>

<constant> ::= <symbol> | <string> | <integer> | <float> | <instance-name>

<variable> ::= <single-vble> | <multi-vble>

<single-vble> ::= ?<symbol>

<multi-vble> ::= $?<symbol>

<CE> ::= <ordered-pattern> | (and <CE>) | (or <CE>) | (not <CE>)

<function-call> ::= (<function-name> <expression>*)

<function-name> ::= = | > | >= | < | <= | <> | eq | neq | <member> ;; just a few examples

<expression> ::= <constant> | <variable> | <function-call>

<term> ::= <single-term> | <multi-term>

<single-term>::= <constant> | <single-vble>

<multi-term>::= <multi-expression> | <multi-vble>

<member> ::= (member <single-term> <multi-term>)

6. Annex: BNF syntax of CLIPS rules (RHS of rules)

<action> ::= (assert <ordered-pattern>) | (retract <term-retract>*) |
(bind <variable> <term-bind>) | <action-multi-value>

<term-retract> ::= <integer> | <single-vble>
<term-bind> ::= <constant> | <variable> | <function-call>
<action-multi-value> ::= (printout …) | (read ….) | (readline ….)