Intelligent Systems: Reasoning and Recognition

James L. Crowley

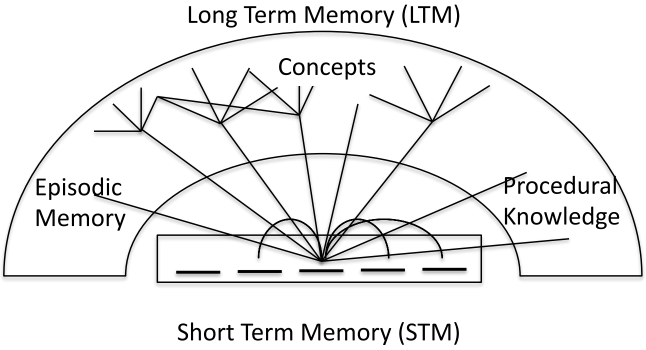
Ensimag 2 Second Semester 2017/2018

Lesson 16 6 April 2018

Declarative Knowledge Representation with the CLIPS

Knowledge Representation in Rule-Based Systems

Most models of human cognition posit some form of "spreading activation" (Anderson 83) in which activation energy associates cognitive "units" in short term memory with concepts, episodes and procedures in long term memory.



Rule-based production systems provide a programmable implementation for this model.

Three techniques are commonly used to represent knowledge in a rule-based production system

An interpreted language - for procedural knowledge
Schema - to represent concepts and frames
Rules (productions) - for activation of LTM from STM

An interpreted language is a programming language in which instructions are interpreted and executed directly at run time, without previous compiling. Interpreted programs permit programs to be treated as data and to be modified dynamically.
The classic interpreted language is LISP. Popular modern languages include Python and Java.

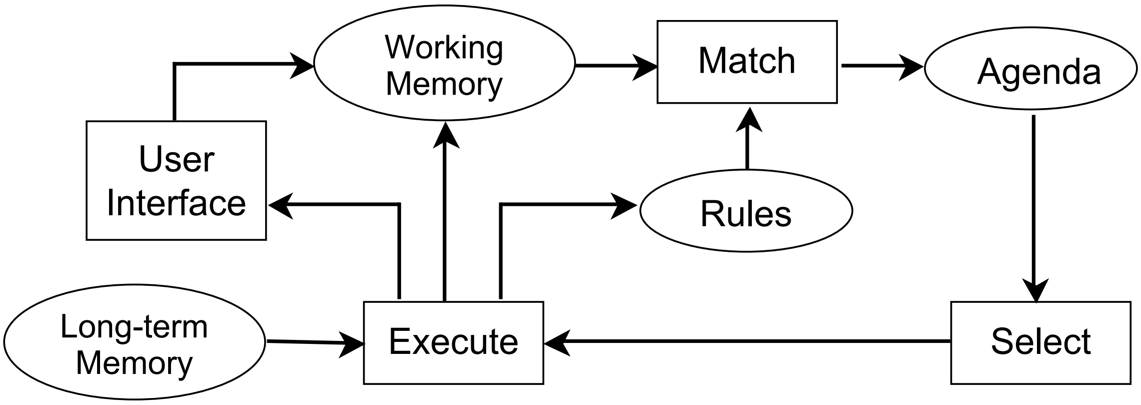
Schema are patterns (or templates) for representing concepts or data. The simplest form of schema is a list of data. A more common form is a named collection of attribute-value pairs in which the attributes (or slot) names act as a key for indexing.

Rules associate concepts in short term and long term memory. Rules take their inspiration from the observation of conditioned reflexes in animals and humans.
Rules are also known as productions.

Production System Architecture

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: Select an activation for execution.

Execute: Execute the actions specified in the activation.

The speed of the inference engine is measured in cycles/second.

The CLIPS Program Interpreter

The CLIPS interpreter interprets instruction in a Lisp-like pre-fix notation:

(operator data*)

Programs are composed of "expressions" enclosed in parentheses. The first symbol after an open parenthesis is an operator. Any remaining symbols are data.

Expressions are data. They can be created dynamically be programs.

Expressions may be composed recursively: (operator (operator (operator data*)))

CLIPS includes a large number of pre-defined operators. These are described in the CLIPS Basic Programming guide.

Any of the predefined operators can be included in the action part of a rule, may be used in a user-defined function, or can be entered directly into the interpreter by the user.

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 foobar)
(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.

```
(defrule calculate-distance
   (point ?x1 ?y1)
   (point ?x2 ?y2)
=>
(assert
   (distance (distance ?x1 ?y1 ?x2 ?y2)))
)
```

Bind, Read and Read-Line

New variables can be defined and assigned with bind: (bind ?x 0).
If the variable has not previously been defined, it is automatically created.
The scope (domain of definition) of the variable is the current rule or objects.

Values may be read from a file or from ttyin by read and readline.
Read will read a single symbol.
Read-line reads all characters to the next carriage return and returns a string.

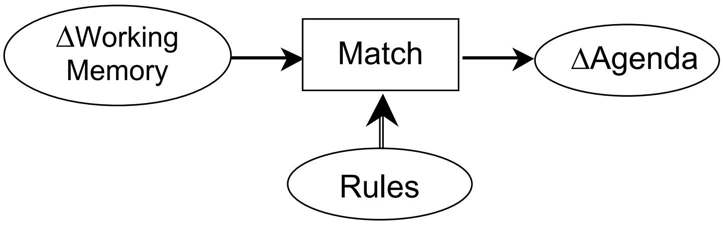
example :

```
(defrule ask-user
   (person)
=>
   (printout t "first name? ")
   (bind ?surname (read))
   (printout t "Family name? ")
   (assert (person ?surname (read)))
   (printout t "Why are you creating this person? ")
   (bind ?string (readline))
)
```

Both Read and Readline are operators.
They must be preceded and terminated by “(“ and “)”.

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.

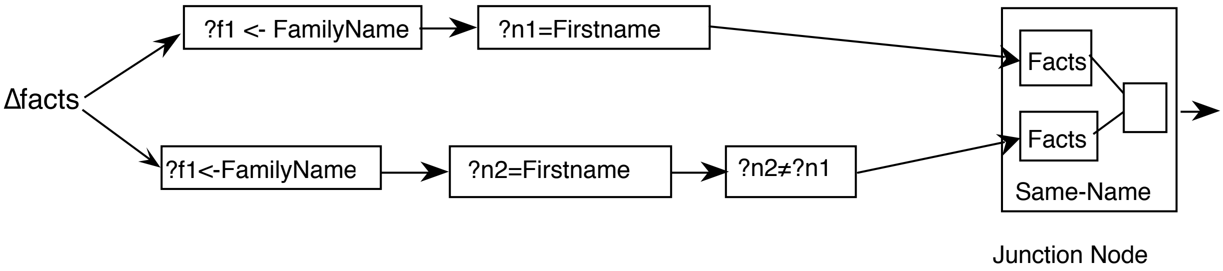
Changes in working memory filter through this decision network generate changes to the agenda.

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

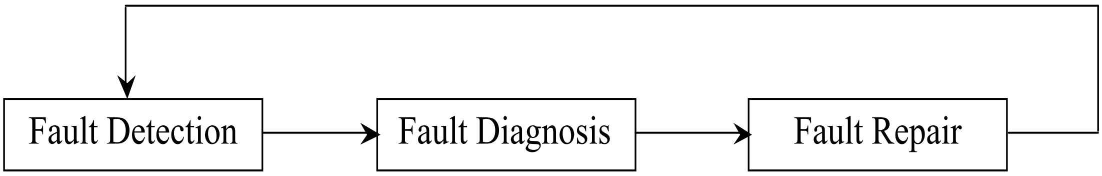
Contexts and Control Elements

A popular programming technique is to organise sets of rules into "contexts". Contexts are indicated by the existence of a token: an element in the facts list that indicates the current context.

(context <Name of the context>)

A classic example is a self-monitoring and self-repair system used for satellites and space applications. Such systems typically operate in cycle with 3 contexts:

Fault-Detection: A set of rules that test the integrity of subsystems
Fault-Diagnosis: A set of rules that determine the origin of an error.
Fault-Repair: A set of rules that reconfigure the system to repair a fault.



A “context” element activates the rules of each context, and de-activates all others. Control rules manage the transitions between contexts. These can be implicit or declarative. For example:

```
(defrule detection-to-diagnosis
    (declare (salience -10))
    ?context <- (context detection)
    (fault ?f detected)
=>
    (retract ?context)
    (assert (context diagnosis))
    (printout t "Fault " ?f " detected!" crlf)
)
```

Each context contains a collection of rules for domain knowledge that diagnosis and suggest a repair for the fault. A salience hierarchy assures transition to the next context after execution of all of the domain knowledge rules.

Declarative Control Structures

An alternative to coding the context transitions in explicit rules, is to encode the context transitions in a declarative data structure and use a single generic transition rule.

```
(deffacts control-list
    (context detection)
    (next-context detection diagnosis)
    (next-context diagnosis repair)
    (next-context repair detection)
)

(defrule context transition rule.
    (declare (salience -10))
    ?P <- (context ?context)
    (next-context ?context ?next)
=>
    (retract ?P)
    (assert (context ?next))
)
```

This is an example of a DECLARATIVE representation of control knowledge.
Declarative structures make it possible to treat knowledge representations as data for calculation. A declarative representation can be used as data to reason about knowledge.

Representing Frames with Objects

Frames are schema with procedures that perform actions to complete slots.
CLIPS provides an object oriented programming system for declaring frames.

CLIPS object classes are declared with a Defclass statement:

BNF:

```
   (defclass <name> [<comment>]
     (is-a <superclass-name>+)
     [<role>]
     [<pattern-match-role>]
     <slot>*
     <handler-documentation>*)
```

example:

```
(defclass PERSON (is-a USER) (role concrete)
   (slot FAMILY (create-accessor read-write))
   (slot FIRST-NAME (create-accessor read-write))
)
```

(1) A (defclass) must have a class name, <class>.
(2) There must be at least one superclass name, <superclass>, that follows the is-a.
(3) A (defclass) has zero or more slots.
(4) Each slot has zero or more **facets**; <facet>, that describe the characteristics of the slot.

An object is an instance of a class, created with “make-instance”.

```
CLIPS> (make-instance of PERSON)
[gen1]
```

In this case the function make-instance generates a new name for the PERSON

```
CLIPS> (make-instance John of PERSON)
[John]
```

In this case, make-instance assigns the name John to the PERSON.

In clips, we can see all the details describing a class with describe-class:

```
(describe-class PERSON)

*********************************************************
Concrete: direct instances of this class can be created.
Reactive:  direct  instances  of  this  class  can  match
defrule patterns.

Direct Superclasses: USER
Inheritance Precedence: PERSON USER OBJECT
Direct Subclasses:
---------------------------------------------------------
SLOTS      : FLD DEF PRP ACC STO MCH SRC VIS CRT OVRD-MSG
SOURCE(S)
FAMILY      : SGL  STC  INH  RW   LCL  RCT  EXC  PRV  RW   put-
FAMILY   PERSON
FIRST-NAME : SGL  STC  INH  RW   LCL  RCT  EXC  PRV  RW   put-
FIRST-NA PERSON

Constraint information for slots:

SLOTS      : SYM STR INN INA EXA FTA INT FLT
FAMILY      :  +    +    +    +    +    +    +    +   RNG:[-
oo..+oo]
FIRST-NAME :  +    +    +    +    +    +    +    +   RNG:[-
oo..+oo]
---------------------------------------------------------
Recognized message-handlers:
init primary in class USER
delete primary in class USER
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
get-FAMILY primary in class PERSON
put-FAMILY primary in class PERSON
get-FIRST-NAME primary in class PERSON
put-FIRST-NAME primary in class PERSON
*********************************************************
```

Class hierarchy

Classes are defined hierarchically using inheritance. Inheritance provides a child class with slots and methods from parent classes.
All user-defined classes are derived to class USER.
Inheritance is described by a "**class precedence list**"

Example:

```
(defclass PERSON (is-a  USER) (slot NAME) (slot FAMILY))
(defclass STUDENT (is-a PERSON) (slot SCHOOL))
(defclass EMPLOYEE (is-a PERSON) (slot EMPLOYER))
(defclass THESARD (is-a STUDENT  EMPLOYEE) (slot THESIS-SUBJECT))
```

Subclasses inherit the slots and methods of parent classes.

CLIPS supports multiple inheritance: classes inherit from multiple super-classes

Abstract and Concrete Classes:

CLIPS classes can be abstract or concrete. Instances may be created only for concrete classes. Abstract classes are only used to define other classes.
By default, classes are "abstract".

To use a class to make an object it must be declared as concrete with a statement.

```
(role concrete)
```

```
(defclass ENSI (is-a STUDENT) (role concrete)
      (slot NATIONALITY (default FRENCH))
      (slot SCHOOL (default ENSIMAG))
)
(make-instance Jean of ENSI (NAME "Jean"))
```

Message Handlers

Slots are accessed (read, write and init) with handlers.

put-<slot>	set the value of a slot
get-<slot>	read the value of a slot
init-<slot>	Initialise the value for a slot

Handlers must be explicitly created using “create-accessor”

```
(defclass PERSON (is-a  USER)
	(slot NAME (create-accessor read-write))
	(slot FAMILY (create-accessor read-write))
)
(defclass STUDENT (is-a PERSON)
	(slot SCHOOL (create-accessor read-write)))
(defclass ENSI (is-a STUDENT) (role concrete)
	(slot NATIONALITY (default FRENCH) (create-accessor read-write))
	(slot SCHOOL (default ENSIMAG) (create-accessor read))
)
```

```
(make-instance Jean of ENSI (NAME "Jean") (NATIONALITY French))
```

Objects can be accessed with send

```
(send [Jean] put-FAMILY "Dupont")
(send [Jean] get-NATIONALITY)
(send [Jean] get-SCHOOL)
```

when (create-accessor read) is declared for a slot, a "get" handler is created.

```
(defmessage-handler <class> get-<slot-name> primary ()
    ?self:<slot-name>)
```

when (create-accessor write) is declared in for a slot, a "put" handler is created.

```
(defmessage-handler <class> put-<slot-name> primary (?value)
    (bind ?self:<slot-name> ?value)
```

or, if this is a multi-slot.

```
(defmessage-handler <class> put-<slot-name> primary ($?value)
    (bind ?self:<slot-name> $?value)
```

Message handlers are used to construct procedures that can complete the slots of a frame.

?Self

consider : (send [OBJ] function) the object [OBJ] is said to be the "active" object.

Within a message handler, the variable ?self provides the address of the active object. This permits directs access to slots and enables calculation.

For example:

```
(defclass THING  (is-a USER) (role concrete)
   (slot NAME (create-accessor read-write) (default A))
)

(defmessage-handler THING ask-name ()
   (?self:NAME)
)
```

?self provides direct access to slots with the notation : ?self:<slot-name>.
This allows access without using the message passing mechanism.

```
(defmessage-handler THING return-name ()
   ?self:NAME)

(defclass THING  (is-a USER) (role concrete)
   (slot NAME (create-accessor read-write) (default A))
    (slot PTR (create-accessor read-write))
)

(make-instance A of THING (NAME A))
(make-instance B of THING (NAME B) (PTR [A]))

(defmessage-handler THING return-name ()
   (send ?self:PTR get-NAME)
)

CLIPS> (send [B] return-name)
A
```

NOTE: You should never need to write:
(bind ?NAME (send ?self get-NAME))
or even (bind ?NAME ?self:NAME)

Use (?self:NAME)

Activating rules with objects

Rules and Classes provide complementary tools for knowledge representation. Objects (class-instances) are not part of the Facts list. However, since version 6 of CLIPS it is possible to activate rules with objects, as if the objects were Facts in working memory.

This is made possible by declaring the class to be "(pattern-match reactive)".
For example :

```
(defclass A (is-a USER)
   (role concrete)
   (pattern-match reactive)
   (slot foo  (create-accessor write)
   )
)

(make-instance a of A)
```

Assertion or retraction of objects of this class are sent to the RETE network.
The matching template is "object", with the template type defined by a default slot: "is-a".

```
(defrule test-for-A
   ?ins <- (object (is-a A))
=>
   (printout t "Object " ?ins " is a member of class A" crlf)
)
```

We can even discover the class name:

```
(defrule test-for-A
   ?ins <- (object (is-a ?A))
=>
   (printout t "Object " ?ins " is a member of class " ?A crlf)
 )
```

The slots of the object are available for condition elements as with templates.

```
(defrule print-A-foo
   ?ins <- (object (is-a A) (foo ?f&~nil))
=>
   (printout t "Object " ?ins " foo = " ?f crlf)
)
(run)
(send [a] put-foo bar)
(run)
```

Thus rules can be used to initialize an object structure.

```
(defclass PERSON (is-a USER) (role concrete)
             (pattern-match reactive)
             (slot FAMILY  (create-accessor read-write))
             (slot NAME  (create-accessor read-write))
             (slot AGR (create-accessor read-write))
             (multislot ADDRESS (create-accessor read-write))
)

(defrule Ask-Family-Names
   ?ins <- (object (is-a PERSON) (FAMILY nil))
=>
   (printout t "What is the family of "?ins "? ")
   (send ?ins put-FAMILY (read))
)

(defrule demande-fname
   ?ins <- (object (is-a PERSON) (NAME nil))
=>
   (printout t "What is the First Name of "?ins "? ")
   (send ?ins put-NAME (read))
)

(make-instance [Fred] of PERSON)
(make-instance [Bob] of PERSON (NAME Bob))
(run)
```

Rules can be applied to objects regardless of their class.
A rule can determine the value of the class from the is-a slot.

```
(defclass STUDENT (is-a USER) (role concrete)
             (pattern-match reactive)
             (slot family  (create-accessor read-write))
             (slot fname  (create-accessor read-write))
             (slot age (create-accessor read-write))
             (slot option (create-accessor read-write))
             (slot promo (create-accessor read-write))
)


(make-instance [Bob] of PERSON (fname Bob) (family Barker) (age
20))
(make-instance  [B] of STUDENT (fname Bob) (family Barker))
```

```
;;
;; Determine the class of an objet
;;

(defrule determine-class
   ?o <- (object (is-a ?c))
=>
  (printout t "The object " ?o " is of class "?c "." crlf)
)
;;
;; Rules can complete values for objects.
;;

(defrule determine-age
   ?o1 <- (object (family ?f&~nil) (fname ?p&~nil) (age
?a&~nil))
   ?o2 <- (object (is-a ?c) (family ?f) (fname ?p) (age nil))
=>
  (send ?o2 put-age ?a)
  (printout t "assign age " ?a " for ")
  (printout t  ?c" " ?p " " ?f "." crlf)
```