Intelligent Systems: Reasoning and Recognition

James L. Crowley

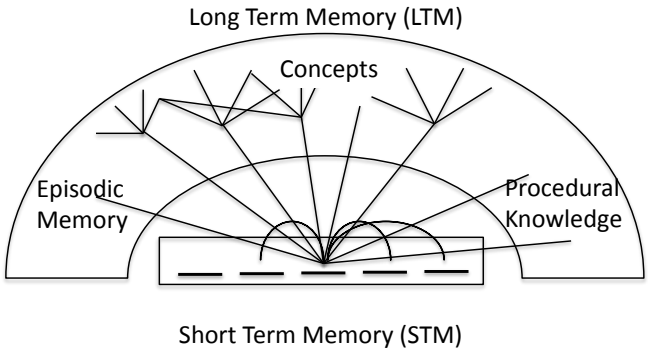
Ensimag 2 Second Semester 2018/2019

Lesson 17 10 April 2019

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.

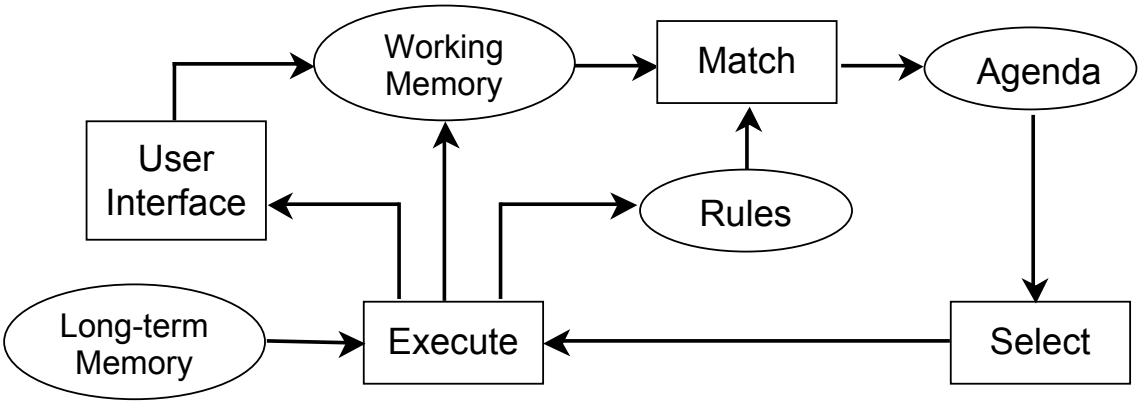
Schema are patterns (or templates) for representing concepts or data. The simplest form of schema is a list of data. A more common form is a named collection of attribute-value pairs in which the attributes (or slot) names act as a key for indexing.

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.

Representing Minsky's Frames with CLIPS 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)
```

Exercise 10 - Family Relations

Exercise 10 10 April 2019

a) Define an abstract class for person with slots NAME (capitalized), father, mother, brother and sister. The slots for brother and sister must be multi-slots so that they can contain a list.

Define a concrete class for MAN as a subclass of person, with the slots "wife" and "gender" having fixed values of "male". Assure that that object of class MAN can activate rules.

Define a concrete class for WOMAN as a subclass of person, with the slots "husband" and "gender" having fixed values of "female". Assure that that object of class MAN can activate rules.

b) Create a rule ask-wife to ask for the name of the wife for a man if the wife is nil. Create a rule ask-husband to ask for the name of the husband for a woman if the husband is nil. Write the rule create-husband and create-wife that will create a MAN or WOMAN for the husband or wife if they do not currently exist.

c) Write the rules ask-father and ask-mother to create the father and mother for each person if these are nil. Do not create the father or mother if the user names them as unknown.

d) define a message handler named get-father-name to get the NAME of the father of a PERSON. If the father is unknown, then print a message stating that the father is unknown.

e) Define a message handler named get-paternal-grandfather to get a ptr to the father of the father of a PERSON. If a father is unknown, then print a message stating that the father is unknown.

f) Define a message handler that return the names of the paternal grandfather for a person

(solution provided during class lecture)