Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 / MoSIG M1 Second Semester 2015/2016

Lesson 17 15 April 2015

Structured Knowledge Representations

Structured Knowledge Representation

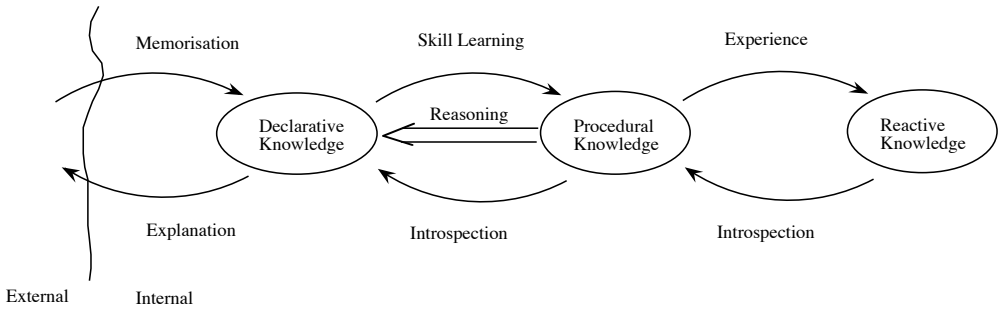
Kinds of Knowledge (reminder)

Cognitive Psychologists identify different categories of knowledge:

Reactive: stimulus - response.

Procedural knowledge: A series actions that lead to a goal. A compiled expression of knowledge, where each action triggers the next action.

Declarative knowledge: A symbolic expression of competence.



Declarative representations are useful for communication and for reasoning about knowledge to generate new knowledge.
Structure Knowledge Representations are a form of Declarative representation.

Structure Knowledge Representations have been explored for many years as a general symbolic representation for declarative knowledge.

As we have seen, a fundamental concept for organizing such structures is the concept of "relation".

Relations as N-Ary Predicates

Relations are a key concept in structured knowledge representations.

Examples include temporal relations, spatial relations, family relations, social relations, administrative organizations, military hierarchies, etc. This is not a closed list.

Relations are formalized as N-Ary Predicates.

Predicates are "truth functions".
A predicates is function that assigns a property to an association of arguments.
The arguments are the "domain" of the function.

Classically, predicates are considered as Boolean functions that can only return a value of TRUE or FALSE.

In probabilistic reasoning, predicates are a probabilistic truth function.
Probabilistic predicates return a number between 0 and 1 representing likelihood according to the axioms of probability.

We will use Predicates to represent and reason about relations.

The "Arity" of a relation is the number of arguments.
Arity represents the number of entities associated by the relation.
Relations may have an arity of 0 or more arguments.

The valence or Arity of a relation is the number of entities that it associates.

Nullary:	Friday()	;; a statement.
Unary:	Man(Bob)	;; a property of Bob
Binary	Brother(Bob, Chris)	;; property of the association of Bob and Chris
Ternary	Triangle(A, B, C)	;; a geometric relation

In some systems it is possible to have functions with variable arity. These are called polyadic functions or Variadic functions.

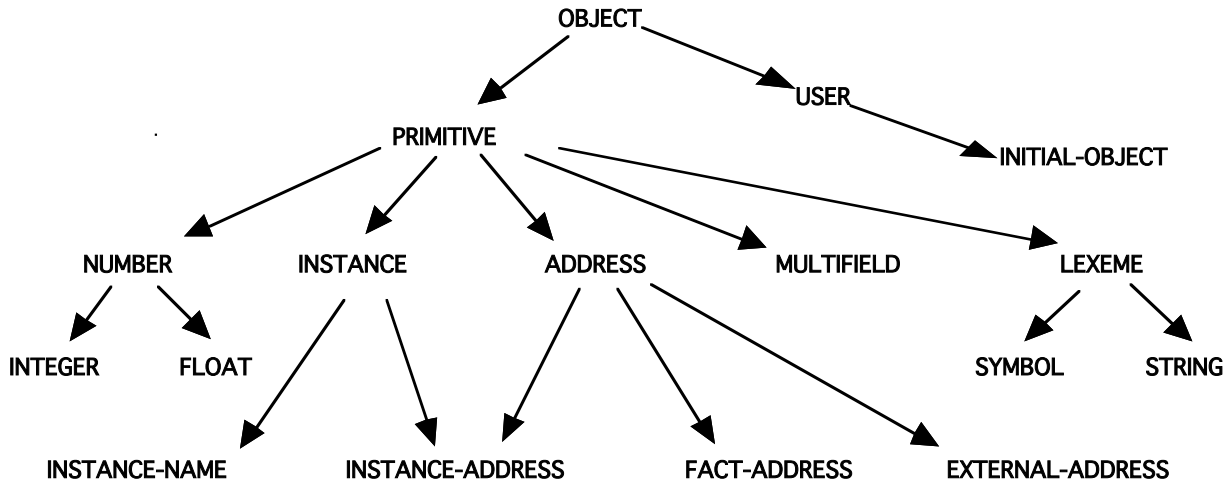
Variadic: Set(A, B, C, D)

A symbol can be defined as a form of triadic relation between 3 entities: a sign, a thing and an agent. The agent interprets the sign to represent the thing.

Structured Knowledge Representation in CLIPS

In CLIPS, structured knowledge are represented by Objects defined with a form of Object Oriented interpreter named "COOL" : Clips Object Oriented Language.
CLIPS is entirely written in COOL.

CLIPS> (list-defclasses)
The predefined types in CLIPS are all members of the superclass "object":



COOLS allows us to represent entities and relations as objects defined in a class hierarchy.

We will use COOL to give examples of structured knowledge.

Declaration of Classes in CLIPS
A class is declared by a "defclass" expression:

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

(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 objects is an instance of a class:

```
(defclass PERSON (is-a USER)
   (slot family)
   (slot age)
   (multislot address)
)

We create an object with Make-instance:

(make-instance of PERSON)

(describe-class PERSON)
```

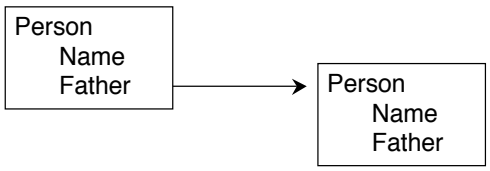
Of course we assign types to slots as with templates.

Representing Relations in CLIPS

Relations can be represented "implicitly" or "explicitly".
In an implicit representation, the thing is represented as a value of a slot. We saw this with the family relations:

(defclass PERSON (slot NAME) (slot FATHER)).

The slot FATHER contains a pointer to an object of the class PERSON that represents the father of the person, The pointer is the object address.



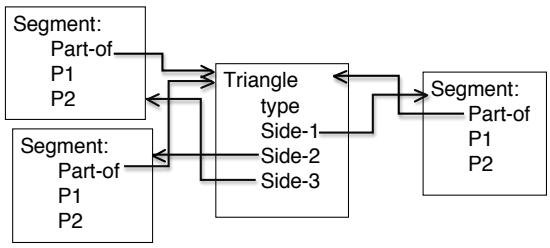
Implicit representations are simple and slightly more efficient in computing and memory.

They are also very limited.

- Implicit representations are static.
- Implicit representations cannot be easily learned or changed.
- Implicit representations can only be used to represent binary or unary relations.

Explicit relations:

In an explicit representation, the relation is represented by a separate object.
The relation object holds the address of each of its entities.



Explicit representations allows a system to annotate the relations with additional properties, and to learn and to reason about relations.

This is an example of the power of representing program as data.
Explicit relations are data that can be used for learning.

More about CLIPS Classes and Objects

BNF:

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

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.

Simple inheritance: A single superclass for each class.
Multiple inheritance: Multiple superclasses for each class.

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

The definition of a slot automatically results in the definition of a set of methods. In CLIPS, for historical reasons, methods are called "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

To obtain handlers we must explicitly create "accessors".

```
(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 Brazil))
```

We access objects with
(send [Jean] put-FAMILY "Dupont")
(send [Jean] get-NATIONALITY)
(send [Jean] get-SCHOOL)

It is possible for the user to create message handlers using the command:

Create-accessor.

The BNF is:
<create-accessor-facet> ::= (create-accessor ?NONE | read | write | read-write)

Syntaxe :

```
(defmessage-handler  <class-name> <message-name>
                     [<handler-type>] [<comment>]
       (<parameter>* [<wildcard-parameter>])
        <action>*
)

<handler-type>   ::= around | before | primary | after
<parameter>          ::= <single-field-variable>
<wildcard-parameter>::= <multifield-variable>
```

Explication

<class-name> :	Name of the class
<message-name> :	Name of the message handler
[handler-type] :	Type of the handler.
<parameters>:	Variable
[wildcard-parameter] :	List variables
<action>:	The set of actions that the object may execute

examples :

when (create-accessor read) is declared in 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)
```

?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)
              (pattern-match reactive)
   )
)

(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)
)

(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 test-foo-eq-toto
   ?ins <- (object (is-a A) (foo ?f&~nil))
=>
```

```
   (printout t "Object " ?ins " foo = " ?f crlf)
)
(run)
(send [a] put-foo toto)
(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))
```

```
;;
;; Determiner la classe d'un objet
;;

(defrule determine-class
   ?o <- (object (is-a ?c))
=>
  (printout t "The object " ?o " is of class "?c "." crlf)
)
;;
;; Completer un objet par un autre
;;

(defrule deduire-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 "Affecter age " ?a " pour ")
  (printout t  ?c" " ?p " " ?f "." crlf)
```

Example : Family Relations

Family relations, such as father, mother, brother and sister can be represented by slots. Answers are determined by "handlers"

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

a) Define a concrete class for MAN as a subclass of person, with the slots "wife" and "gender" having fixed values of "male".

```
(defclass PERSON (is-a USER)
    (pattern-match reactive)
    (slot NAME (create-accessor read-write))
    (slot FATHER (create-accessor read-write) (default unknown))
    (slot MOTHER (create-accessor read-write) (default unknown))
    (multislot BROTHERS (create-accessor read-write))
    (multislot SISTERS (create-accessor read-write))
)
```

b) Define a concrete class for MAN as a subclass of person, with a slot for "wife" and a slot for "gender" having fixed values of "male".

```
(defclass MAN (is-a PERSON)
    (role concrete)
    (slot WIFE (create-accessor read-write))
    (slot GENDER (default MALE)(create-accessor read))
)
```

c) Define a concrete class for WOMAN as a subclass of person, with the slots "husband" and "gender" having fixed values of "Female".

```
(defclass WIFE (is-a PERSON)
    (role concrete)
    (slot HUSBAND (create-accessor read-write))
    (slot GENDER (default FEMALE)(create-accessor read-write))
)
```

d) Create a rule to build the family structure by asking for the wife for a man, and the husband for a wife, and the father and mother for each person.

```
(defrule ask-wife
    ?M <- (object (is-a MAN) (NAME ?n) (wife  nil))
=>
    (printout t "Who is the wife of " ?n "? ")
    (bind ?Wife (read))
    (send ?M put-wife ?Wife)
    (if (neq ?Wife nil) then
        (make-instance ?Wife of WOMAN (NAME ?Wife) (husband ?n)))
)

(make-instance [Jean] of MAN (NAME Jean))
(make-instance [Paul] of MAN (NAME Paul))

(defrule ask-father
    ?M <- (object (is-a MAN) (NAME ?n) (father  unknown))
=>
    (printout t "Who is the father of " ?n "? ")
 (bind ?NAME (read))
    (send ?M put-father ?NAME)
 (make-instance ?NAME of MAN (NAME ?NAME))
)
```

e) Define the message handlers for the class PERSON that can determine the objects that represent the paternal Grandmother and Grandfather.

```
(defmessage-handler PERSON paternal-grandfather ()
    (send ?self:father get-father)
)


(defmessage-handler PERSON paternal-grandfather ()
    (if (neq unknown ?self:father)
    then (send ?self:father get-father)
    else
    (printout t "the father of "?self:NAME "is unknown" CRLF)
    )
)

(defmessage-handler PERSON paternal-grandmother ()
    (send ?self:father get-mother)
)
```