Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 / MoSIG M1 Second Semester 2014/2015

Lesson 8 6 March 2015

Structured Knowledge Representation

Object Oriented Programming

The modern technique of "Object Oriented Programming" co-evolved with the emergence of programming tools for Schemata, with much of the key development happening in the late 60s.

In software engineering, object oriented programming begain with Simula and SmallTalk.

Simula-67 (1967)
SmallTalk (Goldberg, 1973)

OO concepts were easy to program in the interpreted symbolic language "Lisp". A standardized OO system was proposed in the late 70's under the name "Flavors". Flavors proved very useful for structured knowledge representation.

Modern forms include Java and C++

Many of the same concepts were introduced in Artificial Intelligence during the same period.

Semantic networks were proposed as a "pivot language" for natural language understanding and machine translation. (Quinlan 68, Carbonel 68)

Frames were proposed to organize and control computer vision systems (Minski 68)

Knowledge "units" were proposed for reasoning in the KRL system (Bobrow 71).

The vocabulary and programming methods were very similar to object oriented programming.

Structured Knowledge Represenation in CLIPS

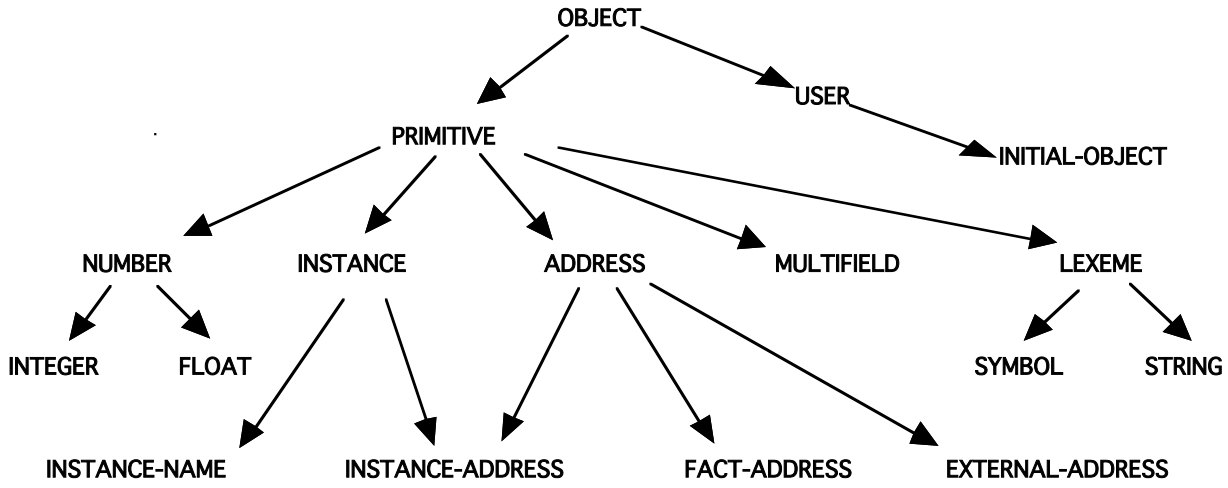
Since CLIPS 4, the clips environment contains an object oriented program tool named "COOL" : Clips Object Oriented Language.

In CLIPS 6, the clips environment was completely rewritten using COOL.

Definition of Classes in CLIPS

CLIPS> (list-defclasses)

The predefined types in CLIPS are all members of the superclass "object":



A class is defined by the 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.

Exemple :

```
(defclass PERSON (is-a USER) (role concrete)
   (pattern-match reactive)
   (slot NAME  (default "John Doe")
            (create-accessor read-write))
   (slot age (create-accessor read-write))
   (multislot address (create-accessor read-write))
)

(defclass ENSI (is-a PERSON) (role concrete)
   (slot year (create-accessor read-write))
   (slot  OPTION  (default  ISI)  (create-accessor  read-
write))
)
```

An objects is an instance of a class:

```
(make-instance <instance> of <class> (<slot> <value>)...)

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

(make-instance of PERSON)

(describe-class PERSON)
```

ROLE:

(role concrete)
Classes may be abstract or concrete. Instances may be created only for concrete classes. Abstract classes are only used to define other classes. Subclasses inherit the slots and methods of parent classes.

```
(make-instance Jim of PERSON (NAME "jim")(age 18))
[Jim]  ; CLIPS returns the name of the instance made.

CLIPS>(send [Jim] get-age)
18
```

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

Inheritance within a class hierarcy

Classes can be defined hierarchically using inheritance. Inheritance provides a child class with slots and methods from parent classes.

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

For example, consider,

```
    (defclass A (is-a USER))
    (defclass B (is-a A))
    (defclass C (is-a A))
    (defclass D (is-a A))

(defclass E (is-a D A USER))
```

<subclass> (is-a <superclass1> <superclass2> ... <superclass-n>)

If there is more than one superclass, the inheritance is multiple.

Inheritance is described by a "**class precedence list**"
The class precedence list gives the priority among the tree of possible parent classes.

Order : Specific -> General

Rules for multiple inheritance in CLIPS (p80 of CLIPS reference manual)

1) A class has higher precedence than any of its superclasses
2) A class specifies the precedence between its direct superclasses.

In heritance is interpreted as a form of depth first search.

Message Passing and CLIPS handlers

Class instances (objects) are encapsulated. Their contents are not directly visible to the external world. The contents of the object's data are accessible by messages sent to communication handlers. In Object Oriented languages these are called "methods".

In CLIPS, methods are called "message-handlers".
This is because there is a "legacy system" called methods that is obsolete but kept in CLIPS for backwards compatibility. This will not concern us in this course.

The general format for a message is:

```
(SEND OBJECT METHOD ARG*)
```

SEND: the key word SEND.
OBJECT: The address of an instance of a class
METHOD: the method of the class that should be executed
ARG* : The arguments for the method

A message is sent to a unique instance. You must have the address to send a message.

example :
(make-instance [Joe] of PERSON (name "joe")(age 32))
(send [Joe] get-name)

A number of message-handlers are defined by default when a class is declared to be concrete. This set includes: Init, print, and delete.

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)

Accessor Facet	Message Handler Crée.
read	get-<slot>
write	put-<slot>
read-write	get-<slot> et put-<slot>

It is possible to declare other message handlers as well as to change the definition of handlers at run time. unsing the command "defmessage-handler".

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
CLIPS>
```

NOTE: You should never need to write:

```
(bind ?NAME (send ?self get-NAME))
```

or even

```
(bind ?NAME ?self:NAME)
```

Use

```
(?self:NAME)
```

Types of Handlers

Handler Type	Class Role	Return
primary	Performs the majority of the work for the message	Yes
before	Does auxiliary work for a message before the primary handler executes	No
after	Does auxiliary work for a message after the primary handler executes	Yes
around	Sets up an environment for the execution of the rest of the handlers	No

The types, before, after and around make it possible to program "demons".

example :

```
(defmessage-handler THING get-NAME after ()
   (printout t "Quack" crlf)
)
```

get-name (primary) still exists.

Example of a demon

```
(defclass THING  (is-a USER) (role concrete)
   (slot NAME (create-accessor read-write) (default A))
   (slot COUNT (create-accessor read-write) (default 0))
)

(defmessage-handler THING get-NAME after ()
   (printout t "Quack" crlf)
)

(defmessage-handler THING get-NAME before ()
   (bind ?self:COUNT (+ ?self:COUNT 1))
   (printout t ?self:NAME " has been read "?self:COUNT " times." crlf)
)
```

Handlers execute a run-time script.
This script can include sending messages to other objects.
As a result, much of the computation activity is hidden from the user.

Handlers in CLIPS are Polymorphic.
The same name may be used for handlers for different classes.
This gives the illusion that the handler is class-independent.
In fact, the exact handler is determined by the class of the destination object.

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 fname  (create-accessor read-write))
             (slot age (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) (fname nil))
=>
   (printout t "What is the First Name of "?ins "? ")
   (send ?ins put-fname (read))
)

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

Exercise : Family Relations

The goal of this exercise is to program a set of classes and message handlers that can respond to questions about the relations within a family. Family relations, such as father, mother, brother and sister are represented by slots. Answers are determined by "handlers"

a) 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.

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

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

b) 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.

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

d) Define the message handlers that return the Names of the paternal grandfather and grandmother.

e) Define a message handler to determine the pointers to the uncles of a person. (brother of father and brothers of mother). Hint : a list can be created with the function create$. Ex : (a b c) <- (create$ a b c)

f) Define a message handler to determine the names of the uncles.

g) Define the message handler to determine the list of names for all of the grandparents.