Systèmes Intelligents : Raisonnement et Reconnaissance

James L. Crowley

Deuxième Année ENSIMAG Deuxième Sémestre 2005/2006

Séance 6 15 mars 2006

Représentation Structurée de la Connaissance

Représentation Structurée de la Connaissance.....2
Schémas.....3
Exemple : Les Relations de Famille.....4
Les Relations.....7

Applications de la Connaissance Structurée ……8
Frames.....8
Scripts.....14
Exemples des scripts.....14
Représentation d'une Hiérarchie des Catégories 16

Techniques Avancés de Programmation en CLIPS..... 18
Evaluation de fonctions par "eval".....18
Definitions dynamique avec a fonction "build".....18
Protocole de Communication avec "eval" et "build" 20
exemple d'execution d'interpret 22
Regle de communication par message 23
Inclusion des fonctions extern dans CLIPS 24

Représentation Structurée de la Connaissance

Intelligence : (Petit Robert)

"La faculté de connaître et comprendre.

On a vu que Connaissance = Compétence.

Qu'est que c'est à comprendre?

Association entre choses perçues et choses connus.

Phénomène : tout ce qui est objet d'expérience possible. ,
Tout ce qui se manifeste par l'intermédiaire des sens.
(ref : Critique du Raison Pure, I. Kant, 1781)

Pour comprendre une scène ou une histoire, il faut trouver une correspondance entre les éléments perçus et les choses connus.

Les perceptions brutes (les phénomènes) sont compris par associé aux catégories mentales (les concepts). Cela exige une représentation. Une telle représentation et fourni par les Schémas

Schémas

Qu'est que c'est à comprendre?

Association entre choses perçues et choses connus.

Pour comprendre une scène ou une histoire, il faut trouver une correspondance Entre les éléments perçus et les éléments qui représentent les connaissances. Les schemas fournissent une représentation déclarative pour la connaissance.

Elle permet de raisonner et comprendre les scènes, les histoires (orales ou écrites), les textes, les idées ou les plans.

Exemple : Les Relations de Famille

Exercice :

L'objet de cet exercice est de réaliser un système de classes permettant de répondre aux questions concernant les relations entre les membres d'une famille. Les réponses sont générées par des "handlers".

a) Définir la classe "personne" avec les slots "nom", "pere", "mere", "freres" et "soeurs" . Les slots frères et soeurs peuvent contenir une liste.

```
(defclass PERSON (is-a USER)
    (role abstract)
    (slot ID (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))
)
```

Définir la classe "homme" (sous-classe de personne) avec le slot "épouse" et le slot "sexe" avec une valeur fixe "masculin".

Définir la classe "femme" (sous-classe de personne) avec le slot "époux" et le slot "sexe" avec une valeur fixe "feminin".

```
;;
;;  Class pour personne, Homme et Femme
;;

(defclass MAN (is-a PERSON)
    (role concrete)(pattern-match reactive)
    (slot wife (create-accessor  read-write) (default unknown))
    (slot sex (storage shared)
            (default male) (create-accessor read))
)

(defclass WOMAN (is-a PERSON)
    (role concrete)(pattern-match reactive)
    (slot husband (create-accessor  read-write) (default unknown))
    (slot sex (storage shared)(access read-only)
            (default female) (create-accessor read))
)
```

Faire la règle pour demander les membres d'une famille

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

```
    (if (neq ?ID nil) then
        (make-instance ?ID of WOMAN (ID ?ID) (husband ?n)))
)

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

(run 1)
(send [Paul] get-wife)

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

b) Définir des "message-handlers" pour la classe PERSON qui déterminent le "gandmere" et le "grandpere" paternels.

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

ou bien

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

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

c) Définir des "message-handlers" pour la classe PERSON qui déterminent les noms des "gandmere" et le "grandpere" paternels.

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

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

d) Définir le message handler pour la classe PERSON qui détermine les oncles (freres du pere et freres de la mere). Aide : On peut composer une liste avec create$. Ex : (a b c) <- (create$ a b c)

```
(defmessage-handler PERSONNE uncles ()
    (create$ (send ?self:father get-brothers)
             (send ?self:mother get-brothers)
 )
)
```

e) Définir le message handler pour la classe PERSON qui détermine les noms des oncles (freres du pere et freres de la mere). Aide : On peut composer une liste avec create$. Ex : (a b c) <- (create$ a b c)

```
(defmessage-handler PERSON name-the-uncles ()
    (bind $?uncles
   (create$ (send ?self:father get-brothers)
            (send ?self:mother get-brothers)
   )
 )
 (progn$ (?uncle $uncles)
     (printout t "the names of " ?oncle " is ")
     (printout t (send ?Uncle get-ID) crlf)
  )
)

(defrule ask-brother
    ?M <- (object (is-a MAN) (ID ?ID) (brothers $?brothers))
    (test (eq (nth 1  $?brothers) unknown))
=>
    (printout t "Who is the brother of " ?ID "? ")
    (bind ?b (read))
    (if  (eq ?b nil) then (bind $?brothers (delete$ $?brothers 1 1))
        else (replace$ $?brothers 1 1 ?b))
    (send ?M put-brothers $?brothers))
)
```

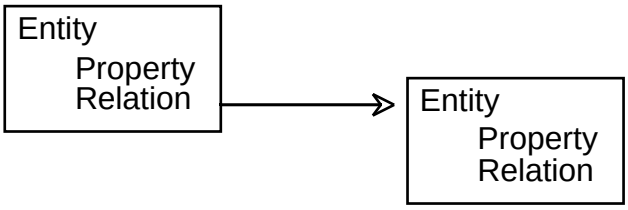
f) Définir le message-handler pour la classe PERSON qui détermine la liste des noms de tous les grands-parents.

```
(defmessage-handler PERSON grand-parents ()
   (create$
      (send (send ?self:father get-father) get-ID)
      (send (send ?self:father get-mother) get-ID)
      (send (send ?self:mother get-father) get-ID)
      (send (send ?self:mother get-mother) get-ID)
   )
)
```

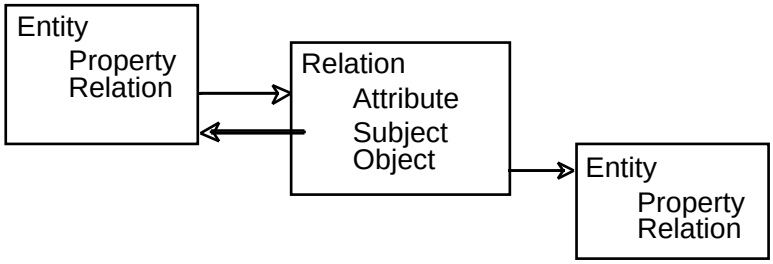
Les Relations

Une relation peut être représenté d'un manière implicite (ex-valeur dans un slot) ou explicite (ex. par un objet de type "relation"). Les relations explicitent peuvent avoir un les attributes ainsi qu'un arity > 1.

Implicite :



Explicite :



L'Arity d'une relation est le nombre de choses associées.

Monadic ou unaire:	male(Jim)
dyadique (ou binaire):	instance-of (Jim, person)
triadique (ou ternaire) :	action (Jim, flew, AF357)

Les relations monadiques servent à représenter les propriétés.
Ils peuvent être représenté par la valeur dans un slot.

La relation dyadique associe deux objets. Ils peuvent être représenté par un pointer dans un slot.
Les relations triadiques exigent un nœud (un objet) avec les pointers pour les rôles.

Exemple : Un symbole est une relation triadique entre une chose, un signe et un agent

```
(defclass symbol (is-a USER)
    (slot signe)
    (slot chose)
    (slot agent)
)
```

Comprendre un symboles voudrais dire trouver les entités pour chaque rôle.

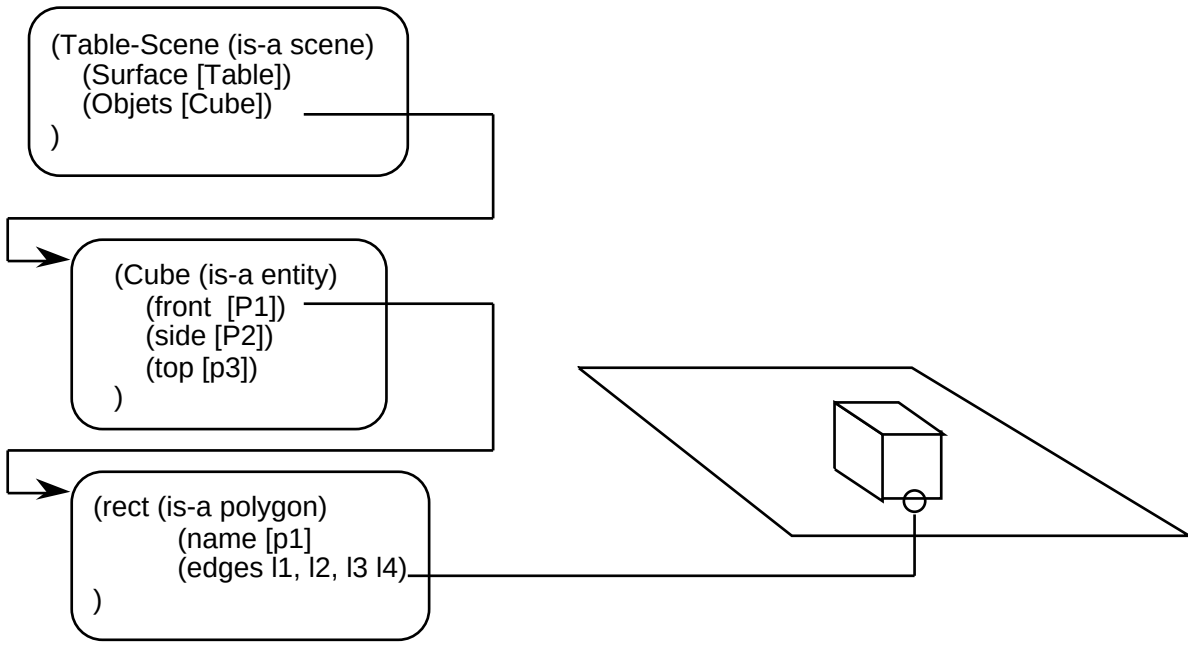
Applications de la Connaissance Structurée

L'expression de la connaissance déclarative par structure de schéma a plusieurs applications. Trois applications classique sont les Frames, les Scripts et les Réseaux Sémantiques.

Frames : Un assemblage des schémas pour décrire une scène ou un contexte.
Script : Un assemblage des schémas pour décrire une séquence des événements
Réseau Sémantique : Un assemblage des schémas pour décrire le sens sémantique d'un texte.

Frames

Le concept de "Frame" était proposé par Minsky pour guider l'interprétation d'une scène en vision par ordinateur. Un Frame guide l'interprétation d'une manière "top-down". Depuis les concepts à été généralisé, et maintenant on trouve le terme frame utilisé en synonyme avec "schéma".



Un frame est une description d'un contexte composé d'une collection de rôles et relations. Ceci donne une description "explicite" du contexte qui peut servir de guider l'interprétation s'une scène, d'une situation ou d'une histoire.

Les Frames spécifie un contexte pour guider l'interprétation dans un domaine. La structure d'une frame permet de faire les inférences par raisonnement par défaut.
Le domaine est composé d'entités.

```
(defclass ENTITY (is-a USER)
   (role abstract)
   (slot ID (create-accessor  read-write))  ;; a unique ID
)
```

Par exemple, dans le monde de blocs, les blocs et le pince sont les entités.

```
(defclass BLOCK (is-a ENTITY)
    (role concrete)
    (slot AKO (allowed-values cube long))
)

;;;AKO - A Kind Of

(defclass pince (is-a ENTITY)
  (role concrete)
)
```

Les schémas servent à représenter les rôles et les relations entre entités qui joue les rôles. Les exemples de relations comprennent les positions relatives, les composants d'une structure, les relations de famille et les relations de temps.

```
(deffacts BLOCK-RELATIONS
    (BLOCK cube A)
    (BLOCK cube B)
    (BLOCK long C)
    (BLOCK long D)
)

(defrule CreateBlocks
    (BLOCK ?kind ?ID)
=>
    (make-instance of BLOCK (ID ?ID) (AKO ?kind)
)
```

Les entités sont lies par les Relations.
Chaque relation est représentée par une liste: (prédicat arg*)

```
(defclass RELATION (is-a USER)
   (role abstract)
   (multislot expression (create-accessor read-write))
    ;; Forme (predicate arg*)
   (slot length (create-accessor read-write))
)
```

Par exemple :

Arity 0 : (HandEmpty)
Unary : (OnTable A)
Binary: (On A B)
Ternary: (Over A B C) ;; Block A is a bridge over B et C.

Voici une règle pour aider à compléter les RELATIONS

```
(defrule assign-length
   ?R <- (object (is-a RELATION) (length nil) (expression $?e))
=>
   (send ?R put-length (length $?e))
)
```

On peut vérifier si une expression est "vraie" (équivalent à relation) avec :

```
(defmessage-handler RELATION verify ($?exp)
  (and    (eq  (length ?self:expression)  (length $?exp))
          (subsetp $?exp ?self:expression)
   )
)
```

On peut générer automatiquement les relations partir des facts par :

```
(deffacts BLOCK-RELATIONS
    (RELATION OnTable A)
    (RELATION OnTable B)
    (RELATION Over C A B)
    (RELATION HandEmpty)
)

(defule MakeRelations
    (RELATION $?R)
=>
    (make-instance of RELATION (expression $?R)
)
```

Une situation est une ensemble de relations entre les entités.
Pour avoir de la généralité, les situations sont defines avec les entities abstrait appelé “roles”.

Rôle Un type abstrait pour une entité.

Exemple : Une tige de bois peut avoir le rôle de baton, ou de pointeur, ou de cale.
Pour jouer un rôle, une entité doit satisfaire une teste. En principe l'ensemble d'entités admissibles pour un rôle est un ensemble ouvert, défini par intention. (Par le test). Cet ensemble peut être fermé, définit par extension (liste d'entités).

Admissibilité : Un prédicat sur une entité qui détermine s’il peut jouer un role.
Sa définition est fondée sur les actions rendus possible par l'entité.

Des roles sont les entités abstrait pour une Frame.

```
(defclass ROLE (is-a USER)
```

```
   (role concrete) (pattern-match reactive)
   (slot ID (create-accessor read-write))
   (slot entity (create-accessor read-write))
   (slot  a-test (create-accessor read-write)
                 (default default-test))
   (slot  what-is (create-accessor read-write))
 )
```

Pour interpréter les entité, il faut évaluer leur admissibilité pour les roles.

```
(defmessage-handler  ROLE  Assign  (?e)
            ;; function to evaluate suitability of entity for role
    (send ?self ?self:a-test ?e))   ;; if a test exists, apply it.


(defmessage-handler ROLE default-test (?e)
   ( return 0)  ;;; par defaut no entitiy is suitably
)

(defclass SITUATION (is-a USER)
   (role concrete) (pattern-match reactive)
   (slot ID  (create-accessor read-write))
   (slot context (create-accessor read-write))
   (multislot roles (create-accessor read-write))
   (multislot relations (create-accessor read-write))
   (multislot neighbors (create-accessor read-write))
)

(defclass REALITY (is-a USER)
      (role concrete) (pattern-match reactive)
       (multislot  relations (create-accessor read-write))
)
```

Pour voir si une relation est vraie dans un contexte:

```
(defmessagehandler REALITY verify (?exp)
(progn$ (?rel self:$?RELATIONS)
    (if (send ?rel ?exp) then (return TRUE))
    (return false)
)
```

La situation est vrai si tous les relations sont vraies

```
(defmessagehandler SITUATION verify (?Reality)
(progn$ (?rel self:$?RELATIONS
    (if (not (send ?sit verify ?rel)) then (return FALSE))
    (return TRUE)
)
```

Neighbors est une liste de situations directement accessible de cette situation.

Une situation est une conjonction de relations. Les situations form une réseau.
À partir de ceci nous pouvons définir les situations et les contextes.

Context : Une ensemble de rôles de relations

Situation : Une affectation des entités aux rôles complétée par les valeurs des relations entre ces entités.

Une Frame est composée d'une liste de "Roles", une liste de "Relations", et une liste de Situations.

```
(defclass CONTEXT (is-a USER)
   (role concrete) (pattern-match reactive)
   (multislot entities (create-accessor read-write))
   (multislot  roles (create-accessor read-write))
   (multislot  relations (create-accessor read-write))
   (multislot  situations (create-accessor read-write))
 )
```

Le Context sert à guider l'interprétation, par spécification des rôles et les situations possible. Pour chaque role, on cherche un entity adapté.

```
(defmessage-handler CONTEXT assign-roles ( )
    (bind ?entity nil) (bind ?suitability 0)
    (bind ?r nil) (bind ?e nil)
    (progn$ ?r  $?self:roles)
      (progn$ ?e   $?self:entities)
         (bind ?s (send ?r a-test  ?e))
          (printout t "suitability of " ?s " for  Role "
                (send ?r get-ID) " is " ?s CRLF)
          (if (< ?s ?suitability) then
              (bind ?suitability ?s) (bind ?entity ?e))
      )
   )
)

(defmessage-handler CONTEXT verify  (?ARG)
        ;; function to verify if a relation is true
   (progn$ (?rel ?self:relations)
        (if (send  ?rel verify ARG) then (return TRUE))
   (return FALSE)
)
```

Pour voir si une relation est vraie dans un contexte:

```
(defmessagehandler SITUATION verify (?exp)
(progn$ (?rel $?RELATIONS)
    (if (send ?rel ?exp) then (return TRUE)
    (return false)
)
```

Scripts

Un Script est une structure de schémas utilisés pour décrire une séquence d'événements. Un scripte est une forme de connaissance déclarative qui peut servir

1) Communiquer, comprendre et raisonner sur une histoire
2) Comprendre ou raisonner sur une situation ou une séquence d'événements
3) raisonner sur les actions
4) Planifier les actions

Un scripte est composée de :

1) Une scène (en lieu et ses composants).
2) Des "props" (Les objets manipulés dans le script).
3) Les acteurs
4) Des actions
6) Les événements : Un changement de situation ou Contexte
7) Les contextes. Une association de rôles et relations.

Le scripte est une structure de contextes. Dans chaque contexte, un ou plusieurs acteurs font les actions. Les acteurs agissent dans les lieus de la scène avec les props. Les contextes peuvent être organisé en séquence, arbre ou réseau.
Le scripte aide à l'interprétation et à la communication par fournissant les éléments primitifs est les méthodes de leur reconnaissance.

En outre, une secrétaire de plateau permet de prédire les actions et événements possibles et les nouveaux contextes qu'on découle.

Exemples des scripts

1) Manger dans une bonne restaurant française

Scène : La (les) salle(s), l'entrée, les tables (comme lieu), la cuisine
Acteurs : Le maître d'Hôtel, Le serveur, la "bus-boy", les clients
Props : le menu la table (comme objet), les chaises, les couteaux, fourchettes et cuillers, la (les) verre(s)
Actions: Entrer, S'asseoir, lire le menu, commander, manger, boire, demander l'addition, payer, partir.
Contextes: Les phases du repas (arriver, s'installer, commander, manger, payer)

La scène, les acteurs, le props, les actions, les contextes sont tous représenté par les schéma.

2) Achat d'une boisson au distributeur automatique

Scène :	devant la machine
Props:	La machine, les pieces de monnaie, la boisson, le gobelet,
Acteur :	acheteur
actions :	1) Sortir tes pieces de monnaie
	2) Payer
	3) Sélectionner la boisson et les options (sucre, crème, etc.)
	4) Recouper la boisson et le monnaie

Représentation d'une Hiérarchie des Catégories

Une hiérarchie est une structure relationnelle entre entités. On peut organiser les hiérarchies entre les individus, les symboles, les concepts ou d'autres entités.

Par exemple, dans un réseau sémantique, les concepts sont organisés en hiérarchies. Le plus bas niveau est composé de mots. Une représentation hiérarchique permet une inférence des propriétés (par défaut) et une économie de représentation.

La relation d'abstraction ENTRE catégories est parfois appelée "AKO" (A Kind of). Nous allons faire une inférence entre membres de d'une hiérarchie par une sorte d'"héritage dynamique".

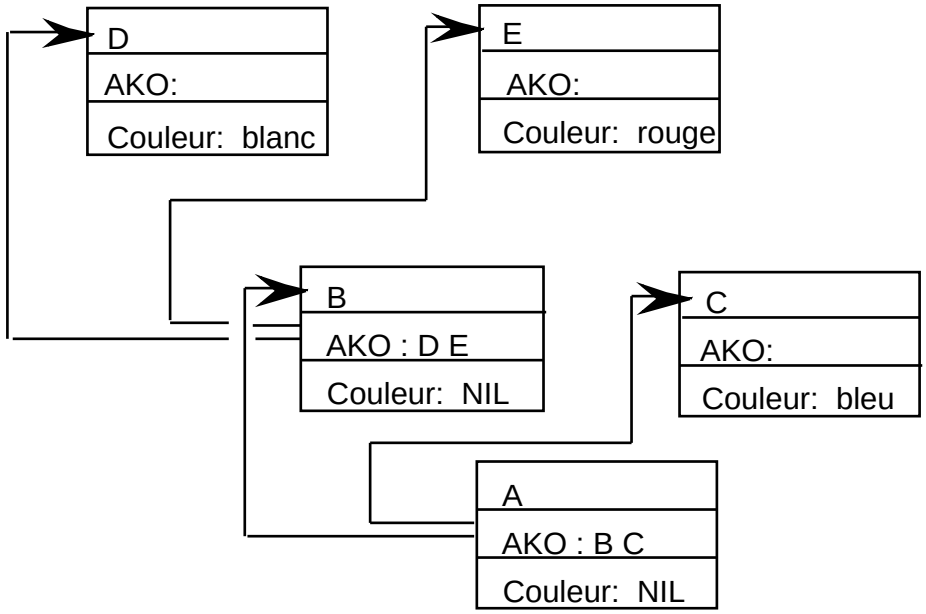
On peut trouver l'héritage simple (Une AKO par concept) ou l'héritage multiple (Plusieurs AKO par classe).

L'héritage dynamique est une forme de parcours de graphe.
Il peut être multiple. Il peut être "strict" ou "defeasible".
Héritage multiple "defeasible" permet les exceptions.

Exemple:

```
(defclass CHOSE (is-a USER) (role concrete)
   (multislot AKO (create-accessor read-write))
   (slot couleur (create-accessor read-write))
)

(make-instance [A] of CHOSE (AKO [B] [C]))
(make-instance [B] of CHOSE (AKO [D] [E]))
(make-instance [C] of CHOSE (couleur bleu))
(make-instance [D] of CHOSE (couleur blanc))
(make-instance [E] of CHOSE (couleur rouge))
```



La Relation AKO est réalisée par une “message-handler”.

```
(defmessage-handler CHOSE inherit-couleur  ()
   (if (neq ?self:couleur nil) then (return ?self:couleur)
   else
      (bind ?couleur)
      (progn$ (?super ?self:AKO)
         (bind ?couleur (send ?super inherit-couleur))
         (if (neq nil ?couleur) then (break))
      )
      (return ?couleur)
   )
)

Une message-handler pour une héritage multiple est

;;
;; héritage multiple. Les valeurs sont aussi affecté aux slots.
;;

(defmessage-handler CHOSE mv-inherit-couleur  ()
   (if (neq ?self:couleur nil) then  (return ?self:couleur)
   else
      (bind $?couleur (create$))
      (progn$ (?super ?self:AKO)
          (bind ?c (send ?super inherit-couleur))
          (if (neq ?c nil) then
            (bind $?couleur (insert$ $?couleur 1 ?c))
          )
      )
      (return $?couleur)
   )
)
```

Techniques Avancés de Programmation en CLIPS

Evaluation de fonctions par "eval"

(eval <string>) ou (eval <symbol_expression>)

La fonction "eval" permet d'interprèter une expression comme si il a été tapé par l'utilisateur.
L'expression peut être une chaine de caractères ou des symboles.

exemple :

```
CLIPS> (eval "(+ 3 4)")
7
CLIPS> (eval "(create$ a b c)")
(a b c)
CLIPS>

;;
;; exemple de eval avec un fonction.
;;

CLIPS> (deffunction f (?a)
         (printout t "f(" ?a ")  = " (random)  crlf))
CLIPS> (f 1)
f(1)  = 16838
CLIPS> (eval "(f 2)")
f(2)  = 5758
CLIPS>
```

Eval n'execute pas un fonction de definitions (ex.: defrule, deftemplate, declasse...)
Pour les definitions, il faut utiliser "build".

Definitions dynamique avec a fonction "build"

(build <string>) ou (build <symbol_expression>)

La fonction build permet l'interpretation d'une chaine de caractere contenant une definition CLIPS , (ex.: defrule, deftemplate, declasse...).

build retourne TRUE si la definition a marché et FALSE s'il y a eu une erreur.

```
;;
;;  exemple de build
;;

CLIPS> (build "(deffunction g (?a) (return ?a)")
TRUE
CLIPS> (g whiz)
whiz

;; qu'est ce passe si on fait ceci?

CLIPS> g(whiz)
g
```

Une Probleme : les guillements dans une chaine de caractères
Il faut les preceder les guillement par \

```
CLIPS> (build "(deffunction f (?a)
   (printout t \"f(\" ?a \") = \" (random) crlf )"
)
TRUE
CLIPS>(f 1)
f(1) = 31051
CLIPS>(eval "(f 1)")
```

Build permet des règles en CLIPS de construire d'autre règles.

exemple :

```
;;
;; build d'un regle
;;

(defrule regle-construit
         (a b c)
        =>
         (printout " (a b c) existe " crlf)
)
```

peut etre construit par :

```
(build "(defrule regle-construit (a b c)
        => (printout t \" (a b c) existe \" crlf) )"
)
```

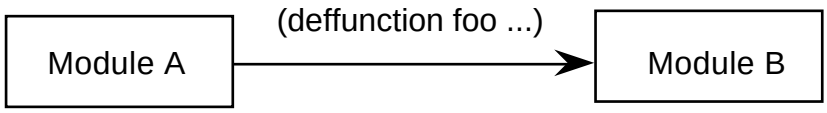
Le chaine de caractère peut être composer par un ensemble de règles.

Protocole de Communication avec "eval" et "build"

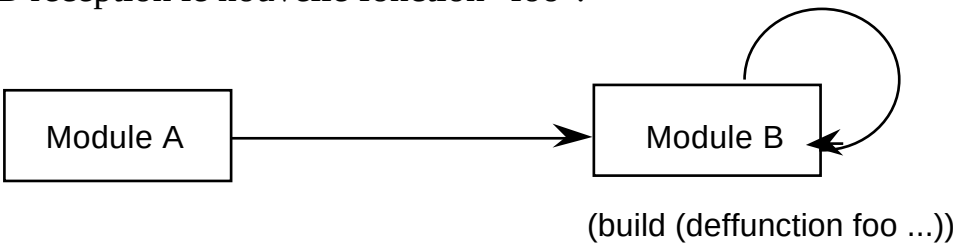
Avec "build" et "eval", Deux processus CLIPS communicant par Socket peut se definir leur protocole de communication dynamiquement.

C.-a-d. Un processus CLIPS peut programmer un autre.

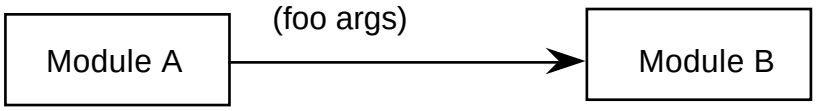
Processus A programme processus B par message



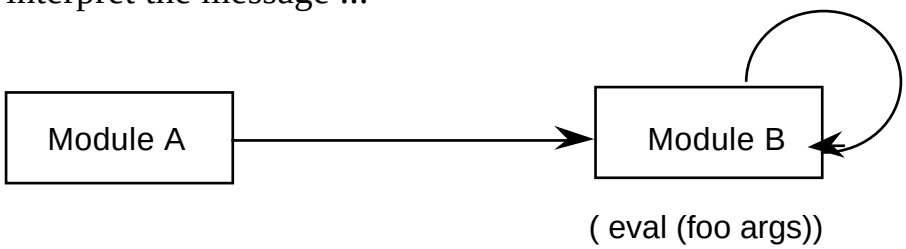
Processus B reception le nouvelle fonction "foo".



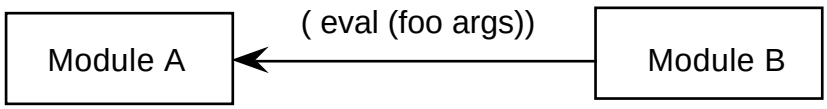
Processus A peut ensuite executer fonction foo par message a Procesus B



Processus B interpret the message ...



.... et retourne un resultat.



Si un message commence par ""(def..." on appel "build" sinon "eval".

Soit les fonction d'envoie et reception "send-mess" et "get-mess".

```
(deffunction interpret ()
   (bind ?mess (get-mess))
   (bind ?sub (sub-string 2 4 ?mess))
   (if (eq ?sub "def")
      then (build ?mess)
      else (eval ?mess))
)
```

On peut verifier que la message est a "interpreter" en testant
si le premiere caractere est un "(".

```
(deffunction interpret ()
   (bind ?mess (get-mess))
   (if (neq (sub-string 1 1 ?mess) "(")
   then  (printout t "message : " ?mess crlf)
         (return TRUE)
   else
      (bind ?sub (sub-string 2 4 ?mess))
      (if (eq ?sub "def")
         then (build ?mess)
         else (eval ?mess))
      )
)
```

On peut simuler send-mess et get-mess avec printout et readline.

```
   ;;  send-mess permet l'envoie d'une message par "socket"
   ;;   on simule send-mess ici avec printout

   (deffunction send_mess (?message) (printout ?message))

   ;; get-mess recoit une message par socket.
   ;;  On simule get-mess avec readline

   (deffunction get-mess ()
               (printout t "Message? ")       ;; interrogation
               (return (readline)))           ;; reponse

   exemple :

   CLIPS> (send-mess "This is a test")
   This is a test
   CLIPS> (get-mess)
   Message? this is a test
   "this is a test"
   CLIPS>
```

On peut interpreter les message par la fonction "interpret".

```
(deffunction interpret ()
  (eval (get-mess))
)
```

Exemple :

```
CLIPS> (deffunction send-mess (?message) (printout t ?message
crlf))
CLIPS> (deffunction get-mess () (printout t "Message? ") (return
(readline)))
)

CLIPS> (deffunction interpret () (eval (get-mess)))

CLIPS>  (interpret)
Message? (assert (a b c))
<Fact-0>
CLIPS> (facts)
f-0     (a b c)
For a total of 1 fact.
CLIPS>
```

exemple d'execution d'interpret

```
CLIPS> (interpret)
Message? this is a test
message : this is a test
CLIPS> (interpret)
Message? (assert (this is a test))
<Fact-0>
CLIPS> (facts)
f-0     (this is a test)
For a total of 1 fact.
CLIPS> (interpret)
Message? (deftemplate A (slot b) (slot c))
TRUE
CLIPS> (interpret)
Message? (assert (A (b "test")))
<Fact-1>
CLIPS> (facts)
f-0     (this is a test)
f-1     (A (b "test") (c nil))
For a total of 2 facts.
CLIPS>
```

Regle de communication par message

On peut avoir une regle qui recoit et interpret les messages :

```
(defrule get-message
   ?go <- (go)
   (not (stop))
=>
  (retract ?go) (assert (go))
  (send-mess (interpret))
)
```

```
CLIPS> (run)
Message? this is a test
message : this is a test
TRUE
Message? (assert (this is a test))
<Fact-19>
Message? (deffunction stop () (assert (stop)))
TRUE
Message? (deffunction f1 (?x ?y) (+ ?x ?y))
TRUE
Message? (f1 1 2)
3
Message? (deffunction f2 (?x ?y) (* ?x ?y))
TRUE
Message? (f2 2 3)
6
Message? (stop)
<Fact-10>
CLIPS>
```

Inclusion des fonctions extern dans CLIPS

CLIPS est un système ouvert ("Open System" en anglais)
Il est écrit en C et distribué avec sa code source.
Ceci rend facil l'inclusion de fonctions extern, écrit par l'utilisateur,
dans l'environnement de l'interpret.

L'extension de CLIPS est decrit dans le manual :
Vol 2 : Advanced Programming Guide

Pour ajouter une fonction extern il faut
1) compiler le fonction.
2) Ajouter une declaration de la fonction et ses parametres dans main.c
3) Inclure son code objet (fonction.o) dans le "make-file"
et refaire le "make".

Les fonctions sont declarés avec DefineFunction ou DefineFunction2

```
int  DefineFunction(functionName,functionType,
                    functionPointer,actualFunctionName);

int  DefineFunction2(functionName,functionType,
                             functionPointer,actualFunctionName,
                             functionRestrictions);

char *functionName, functionType, *actualFunctionName;
int    (*functionPointer)();
char    *functionRestrictions
```

Exemple des declarations :

```
UserFunctions()
{
   /* Declare your C functions if necessary. */

   extern double ma_fonc();
   extern VOID *dummy();

/* Call DefineFunction to register user-defined functions. */

   DefineFunction("fonc",'d',ma_fonc,"ma_fonc");

}
```

```
1
```

Les paramètres de DefineFonction sont

1) La nom à utiliser dans l'interpreter CLIPS.
   Pas forcement le meme que le nom du fonction ecrite en "c".
2) Le type de resultat rendu par le fonction. (ici un double).
3) L'adresse de la fonction.
4) Une representation en chaine de caractere (STRING) de l'adresse de la fonction.

Il est possible de declarer la fonction avec ses arguments en utilisant "DefineFunction2".

```
int      DefineFunction2(functionName,functionType,
                         functionPointer,actualFunctionName,
                         functionRestrictions);

char    *functionName, functionType, *actualFunctionName;
int    (*functionPointer)();
char    *functionRestrictions
```

Les premieres 4 arguments sont les memes que DefineFonction.
Le cinquième est une chaine de caracteres decrivant les arguments.
Cette chaine a le format :
   <min-args> <max-args> [<default-type> <types>*]

Explications : <min-args> <max-args> : des entiers ou "*"

Exemples :
La communication entre deux processus CLIPS sur deux machines differents peut etre assurer par les procedures en C : transmet() et recoit().

La procedure "transmet()" permet de transmetre une chaine "message".
On veut que sa nom en CLIPS soit "send_mess".
Il return un code d'erreur ou NULL si le message est transmit.

La procedure "recoit()" permet de recoit une chaine "message".
On veut que sa nom en CLIPS soit "get_mess".
Il return une chaine ou une code d'erreur.

```
int transmet(char * message; )
{...
}

char *  recoit()
{...
}
```

Dans main.c :

```
DefineFunction2(send-mess, 'i', transmet, "transmet", "11s");
DefineFunction(get-mess, 's', recoit, "recoit");
```

Les codes des types de retour sont :

Return Code	Return Type Expected
a	External Address
b	Boolean
c	Character
d	Double Precision Float
f	Single Precision Float
i	Integer
j	Unknown Data Type (Symbol, String, or Instance Name Expected)
k	Unknown Data Type (Symbol or String Expected)
l	Long Integer
m	Multifield
n	Unknown Data Type (Integer or Float Expected)
o	Instance Name
s	String
u	Unknown Data Type (Any Type Expected)
v	Void—No Return Value
w	Symbol
x	Instance Address

Des codes de types pour les paramètres sont.

Type Code	Allowed Types
a	External Address
d	Float
e	Instance Address, Instance Name, or Symbol
f	Float
g	Integer, Float, or Symbol
h	Instance Address, Instance Name, Fact Address, Integer, or Symbol
i	Integer
j	Symbol, String, or Instance Name
k	Symbol or String
l	Integer
m	Multifield
n	Integer or Float
o	Instance Name

p	Instance Name or Symbol
q	Symbol, String, or Multifield
s	String
u	Any Data Type
w	Symbol
x	Instance Address
y	Fact Address
z	Fact address, Integer, or Symbol