Systèmes Intelligents : Raisonnement et Reconnaissance

James L. Crowley

Deuxième Année ENSIMAG Deuxième Sémestre 2006/2007

Séance 3 14 février 2007

Systèmes de Productions : CLIPS 6.0

Systèmes de Productions

Trois techniques sont utilisées pour représenter la connaissance :

Les règles :
Les schémas :
La logique.

Souvent les systèmes contiennent deux ou même trois formes.

Les règles :

si <CONDITIONS> alors <ACTIONS>
ou si <CONDITIONS> alors <CONCLUSIONS> do <ACTIONS>

Dans leur formes la plus simples, les règles encodent les associations et les lois causales.

Pour l' I. A., les règles sont une technique pour fournir une représentation des connaissances pour le raisonnement.

La programmation par règles est inspiré par les "Réflexes Conditionnés" observées chez les animaux et chez les hommes.

Réflexes Conditionnés : Condition Réaction

Une réaction peut être une affirmation d'un fait ou d'une action à entreprendre.
Les faits sont exprimés sous forme de liste, de structure ou d'objet.

L'association des faits et règles sont faits par un algorithme de
"mise en correspondance".

La présence de variables dans les règles rendre complexe (et donc chèr)
ce calcul de mise en correspondance. Il existe un algorithme "rapide"
pour cette mise en correspondance : L'algorithme RETE.

Langages de programmation:
OPS-5 (né '78) => ART (né '80) => CLIPS (né '85) => CLIPS 6 (né '94)

CLIPS : “C Language Integrated Production System”

L'origine de CLIPS (CLIPS 1.0) est une version "C" du système "ART" (Automated Reasoning Tool) réalisé par la NASA et distribué "gratuitement" (jusqu'à version 5.0). CLIPS, ART et OPS-5 sont basés sur l'algorithme RETE.

Depuis, CLIPS a évolué à inclure la programmation orienté objet et un système de maintenance de vérité. (CLIPS 6.0).

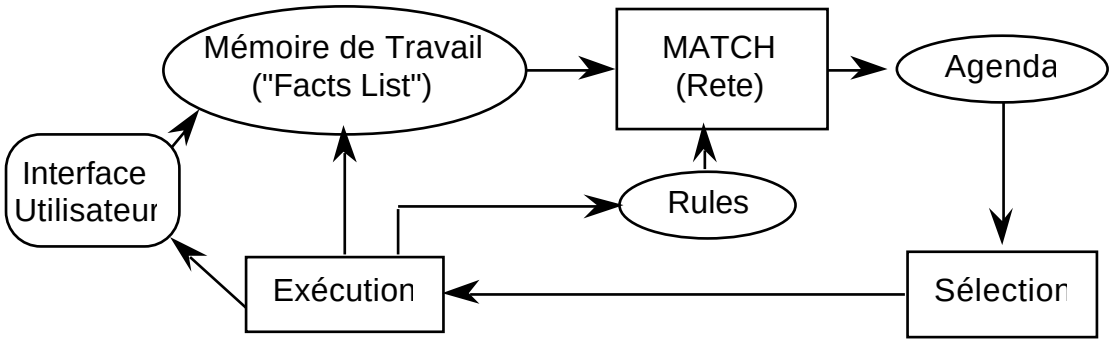
CLIPS est un système ouvert. Il est distribué avec sa source.
Il est facile d'ajouter les procédures écrit en C.

CLIPS est le noyau pour plusieurs milliers de Systèmes Expert.

à la NASA, CLIPS est utilisé afin de faire des systèmes experts pour

1) Contrôle des processus
2) Diagnostic des Pannes.
3) Planification des Missions.

Architecture d'un Système de Production



Le cycle "Recognize-Act" est la base du moteur d'inférence.
Il comporte 3 phases :

Correspondance	(appariement des faits et règles)
Sélection	(Sélection d’une “activation” pour exécution)
Exécution	(L’exécution d’une règle peut engendrer la modification de la mémoire de travail).

Chaque fait est identifié par un "Indice". (ou Estampille ou "Recency")

Dans chaque cycle, toutes les règles sont "mises en correspondance"
avec toute la mémoire de travail.

Les associations de règles et faits sont stockées dans l'agenda.
Une des associations est choisie pour l'exécution.
Il existe plusieurs modes de triage de l'agenda.
Par défaut, l'agenda est une pile (LIFO)

La Liste de Faits

La mémoire de travail de CLIPS est la "FACTS LIST"

Un fait peut être
- Une liste de "champs"
- Une structure composée d'un nom, et d'une liste de paire : (attribut valeur)
- Un instance d'un objet.

Dans la version 6.0 il y a un langage a objet : COOL.
Nous allons utiliser COOL dans les dernières séances du cours
Ici nous utilisons les listes et les stuctures.

Les faits sont créés et détruits par "assert", "retract", "clear" et "reset".

Affirmation d'un fait : (assert <<FAIT>>)
Négation d'un fait : (retract <<FAIT>>)
Vider la Facts List : (clear)
initialisation des listes des faits : (reset)

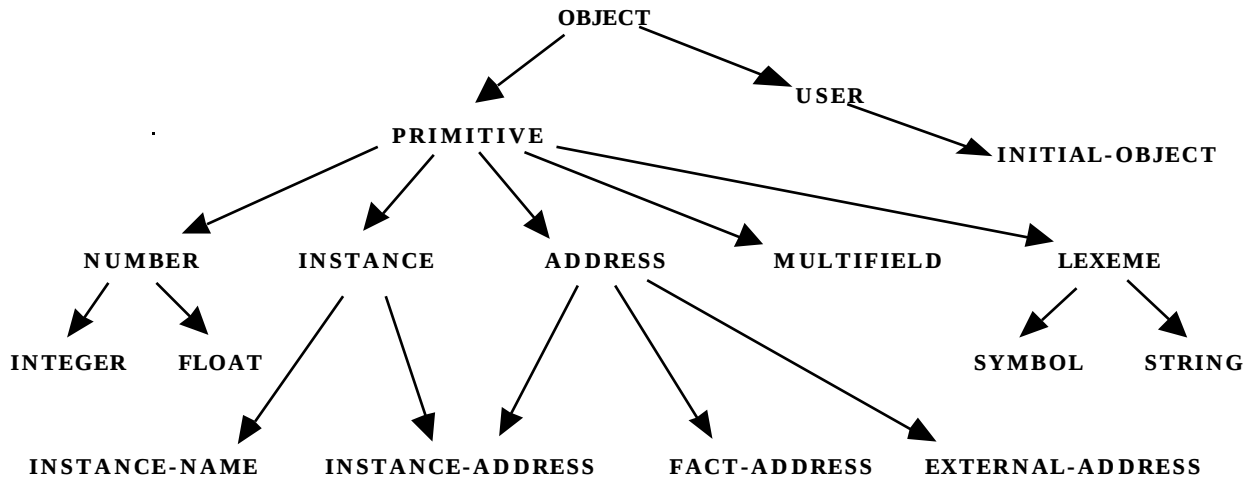
f-0 (initial-fact)

Il y a cinq mots clés à ne pas utiliser dans le premier champ:
mots clés : test, and, or, not, declare

Chaque fait est identifié par un "indice". C'est un nombre unique.
L'indice joue un rôle fondamental dans la phase sélection.

Les types prédefinis en clips

Il y a huit types de données : symbol, string, float, integer, external-address, fact-address, instance-name, et instance-address;
Ils sont definis par une arbre :



Deftemplates

Les structures sont définies par un “template”
Les “templates” permettent de spécifier les faits “non-ordonnés” dans les règles
Une template est composée de :

- un nom
- des champs ("slots")
- des valeurs par défauts.

Les types sont : MULTIFIELD, EXTERNAL-ADDRESS,
LEXEME, SYMBOL, STRING, NUMBER, INTEGER, FLOAT

Exemple:

```
(deftemplate person          ;  une relation pour un person
   "record pour une person"    ;  commentaire optionnel
   (slot name                  ; nom du person
      (default "Pierre Dupont"))  ; Par défaut
   (slot age                   ; age du person
      (default -1))            ; si on ne sait pas
)
```

On peut créer une instance par :

```
(assert (person))
(assert (person (name "joe")))
(assert (person (name "jim") (age 39)))

CLIPS> (facts)
f-0     (person (name "Pierre Dupont") (age -1))
f-1     (person (name "joe") (age -1))
f-2     (person (name "jim") (age 39))
For a total of 3 facts.
```

Commandes pour les templates :

(list-deftemplates)
(ppdeftemplate <template-name>)
(undeftemplate <template-name>)

exemples:

```
CLIPS> (list-deftemplates)
person
For a total of 1 deftemplate.
CLIPS> (ppdeftemplate person)
(deftemplate person "record pour une person"
   (slot name (default "Pierre Dupont"))
   (slot age ((default -1)))
CLIPS> (undeftemplate person)
```

On peut définir les "types" et valeurs par défaut.
On peut fournir les symboles possible avec "allowed-symbols"

exemple :

```
(deftemplate person         ;  une relation pour un person
     "record pour une person"   ;  commentaire optionnel
    (slot name              ; nom du person
        (type STRING)
        (default "Pierre Dupont"))  ; Par défaut
    (slot age            ; masculin ou féminine
        (type NUMBER)
        (default -1)     ; si on ne sait pas
        (range -1 127))  ; declare les valeurs possible
 (slot metier      ; sa métier
        (type SYMBOL)
        (allowed-values artiste ingenieur comercant)
        (default comercant)
 )
)
```

Pour les nombres on peut définir le "range".

```
(slot age
    (type NUMBER)
    (range -1 120))
```

deffacts

Un ensemble de faits peut être associé par une expression "deffacts"

(deffacts <NOM> ["<commentaire>"]
[(<<FAIT-1>>) (<<FAIT-2>>) ... (<<FAIT-N>>))

exemples :
1) les faits "initial-facts" sont affirmés par un "reset"

```
(deftemplate emplacement
    (slot nom (type SYMBOL) (default NIL))
    (slot x (type NUMBER) (default -1))
    (slot y (type NUMBER)(default -1))
    (multislot voisins (default NIL))
)

(deffacts reseaux-d-emplacements
    (emplacement (nom A) (x 0) (y 0) (voisins B C))
    (emplacement (nom B) (x 0) (y 1) (voisins A D))
    (emplacement (nom C) (x 1) (y 0) (voisins A I))
)
```

2) Les autres faits sont aussi affirmé par un "reset"

```
(deffact faits-de-nuit
    "Ceux-ci sont vrai le nuit"
    (soleil couché)
    (étoiles visibles)
)
```

Les Règles en CLIPS

Syntaxe du "defrule"

(defrule <rule-name> [<comment>]
    [<declaration>] ; Rule Properties
    <conditional-element>* ; Left-Hand Side (LHS)
=>
    <action>*) ; Right-Hand Side (RHS)

Si la règle <rule-name> n'existe pas, il est déclaré.
Si un règle avec <rule-name> existe, il est remplacé par defrule.
Il n'y a pas de limites de nombre de conditions ou nombres d'actions.
Les actions sont exécutées d'un manier séquentiel.

S'il n'y a pas de "conditional-element", "(initial-fact)" est pris par défaut

```
<conditional-element> ::= <pattern-CE> |
                          <assigned-pattern-CE> |
                          <not-CE> |
                          <and-CE> |
                          <or-CE> |
                          <logical-CE> |
                          <test-CE>
```

Une Condition (CE) peut être une liste des attributs or un Template.

Liste : (<constant-1> ... <constant-n>)
Deftemplate :

```
(<deftemplate-name> (<slot-name-1> <constant-1>)
                                   .
                                   .
                                   .
                    (<slot-name-n> <constant-n>))
```

La condition peut être un littéral (constant) ou ce peut contenir les variables.

Les VARIABLES

Il y en a deux sortes :

Variable d'indice: Contient l'indice d'une élément de MT
Variable d'attribut : Contient la valeur d'un attribut.

Variables d'indice :

Les variables d'indice sert à récupérer l'indice d'un fait
Ceci permet de retirer ou modifier le fait:

```
(defrule rule-A
   ?f <- (a)
=>
   (printout t "Retracting  " ?f  crlf)
   (retract ?f)
)

(deftemplate A (slot B (default 0)))

(defrule rule-A
   ?f <- (A (B 0))
=>
   (printout t "Changing  " ?f  crlf)
   (modify ?f (B 1))
)
```

Variables d'attribut :

Les variables d'attribut servent à :

1) Récupérer les attributs pour les actions
2) Mettre en correspondance les faits.

Syntaxe :

? - Match avec un item. Valeur non gardée.

$? - Match avec une suite d'items. Les valeurs ne sont pas gardées

?<NOM> - une variable pour un item. Valeur affectée à ?<NOM>
NOM doit être un "mot" (chaîne de caractères entre espaces).

$?<LISTE> - une variable pour une liste d'items.

Exemples :

```
(assert (a b c))
(assert (a b c d e f))
(assert (d e f))

(defrule traiter-1-sur-3
  (a ?x  ?)
=>
(printout t "x = " $?x crlf)
)

(defrule traiter-une-liste
  (a $?x)
=>
(printout t "La liste est " $?x crlf)
)
```

Technique CLIPS de traiter chaque élément d'une liste :

```
(defrule traiter-chaque-element
  (a $? ?x $?)
=>
(printout t "x = " ?x crlf)
)
```

Exemple d'usage d'un variable :

```
((deftemplate person        ;  une relation pour un person
    "record pour une person"    ;  commentaire optionnel
    (slot famille            ; nom du person
       (type STRING)     ; Type chaine de caractères
       (default " Dupont")) ; Par défaut
    (slot prenom             ; nom du person
       (type STRING)     ; Type chaine de caractères
       (default "Pierre ")) ; Par défaut
    )

(defrule Find-same-name
    ?P1 <- (person (nom ?n1) (prenom ?b))
    ?P2 <- (person (nom ?n2) (prenom ?b))
=>
    (printout t ?B ?n1 " et  " ?B ?n2 "Ont le meme prenom"
crlf)
)
```

Syntaxes des Règles - Constraints

On peut imposer les contraints sur les correspondances :

Il y a deux classes de contraintes : Logique et prédicative

Logique : On peut composer les formes avec "et", "ou", "~" (négation)

<v1> ou <v2> - valeur v1 ou valeur v2.

Avec variables, Les variables sont affectées. Il faut que toutes les affections sont le mème pour qu'une règles soit éxutable.

exemple :

(?x & vert | bleu) - CONDITION satisfaite si ?x est vert ou bleu.
(?x & ~rouge) - Condition satisfaite si ?x ≠ rouge

exemples de régles :

```
(defrule test3
  (couleur ?x & vert | bleu)
=>
  (assert (ok))
)

(assert (couleur vert))
(assert (couleur rouge))

(defrule feu-rouge
      (couleur ?x&~vert&~jaune)
=>
      (assert (il faut arreter))
     (printout t "arret" crlf)
)
```

Prédicates

Les affectations peuvent être contraintes par des prédicats.
La variable est suivi par une opérateur logique et une ":"

(?x&:(<predicate> <<arguments>>) variable et prédicats
(?x|: (<predicate> <<arguments>>) variable ou prédicats
(?x&~(<predicate> <<arguments>>) variable et négation d'un prédicats

Prédicats prédefinis :

(numberp <arg>) - vrai si <arg> est un nombre
(stringp <arg>) - vrai si <arg> est une chaîne
(wordp <arg>) - vrai si <arg> est un mot

```
(defrule example-1
  (data ?x&:(numberp ?x))
  =>)

(defrule example-2
  (data ?x&~:(symbolp ?x))
  =>)

(defrule example-3
  (data ?x&:(numberp ?x)&:(oddp ?x))
  =>)

(defrule example-4
  (data ?y)
  (data ?x&:(> ?x ?y))
  =>)

(defrule example-5
  (data $?x&:(> (length$ ?x) 2))
  =>)
```

Fonctions

Une CE peut dépend de la calcul d'un fonction.
Ceci peut être un des fonctions prédefini en CLIPS,
ou une fonction défini par l'utilisateur :
Les fonctions peut être dans le conditions, les actions, ou interprété par l'interpréter.

Une fonction dans une condition est exécuter par la commande "test".

Syntaxe : (test (<fonction> [<<args>>]))

Il existe plusieurs classes des fonctions prédefinit.

Fonctions Logiques :

Fonction	Symbole
négation	not
conjonction	and
disjonction	or

Fonction de comparisons

Fonction	Symbole	exemple
égalité numérique	=	(test (= ?x ?y))
équivalence	eq	(test (eq ?nom ?mere))
inéquivalence num.	!=	(test (!= ?x ?y))
inéquivalence	neq	(test (neq ?nom ?mere))
Supérieur	>	(test (> ?x ?y))
Supérieur ou eq	>=	(test (>= ?x ?y))
inférieur	<	(test (< ?x ?y))
Inférieur ou eq	<=	(test (<= ?x ?y))

Arithmétique :

division	/	(test (< ?x (/ ?y 2)))
multiplication	*	(test (< ?x (* ?y 2)))
addition	+	(test (> (+ ?y 1) ?max))
subtraction	-	(test (< (- ?y 1) ?min))

Deffunctions

L'utilisateur peut écrire les fonctions avec le commande "deffunction".
Une fonction défini par l'utilisateur doit rendre une chaîne, un mot, ou un nombre.

Syntaxe :

```
(deffunction <name> [<comment>]
    (<regular-parameter>* [<wildcard-parameter>])
    <action>*)

<regular-parameter>  ::= <single-field-variable>
<wildcard-parameter> ::= <multifield-variable>
```

exemples :

```
(deffunction ma-fonction (?x)
    (printout t "L'argument est " ?x crlf)
)

(ma-fonction fou)

(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)))
)
```

Les Actions

Dans les actions, une fonction externe est executé par :

(<fonction> <<args>>)

exemple :

```
(deffunction distance (?x1 ?y1 ?x2 ?y2)
  (bind  ?dx (- ?x1 ?x2))
  (bind  ?dy (- ?y1 ?y2))
  (sqrt (+ (* ?dx ?dx) (* ?dy ?dy)))
)

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

Fonctions Prédefinit utilisé dans les actions :

bind - affecter une valeur à une variable.

read, readline - entre les donnée

exemple :

```
(defrule ask-user
    (person)
=>
    (printout t "Prenom? ")
    (bind ?prenom (read))
    (printout t "Nom de famille? ")
    (assert (person ?prenom =(read)))
)
```

Nota : avec bind, les "()" signale une fonction.

avec "assert" il faut un "=" de declencher l'execution.

Les actions Prédefinis

1) assert : Les fait sont entré par l'action "ASSERT"

Syntax : (assert (<<fait>>) [(<<faits>>)])

```
(defrule  j'existe
   (je pense)
=>
   (assert (j'existe!))
)
```

2) retract - pour effacer une fait

```
(defrule je-n'existe-pas
   ?moi <- (je ne pense pas)
=>
   (retract ?moi)
)
```

3) Str-assert Affirmer un chaîne comme une liste

```
(defrule j-existe-je-pense
    (je pense)
=>
    (str-assert "Je pense que j'existe")
)
(facts)

f-0   (Je pense que j'existe)
```

4) Halt : Termination d'exécution