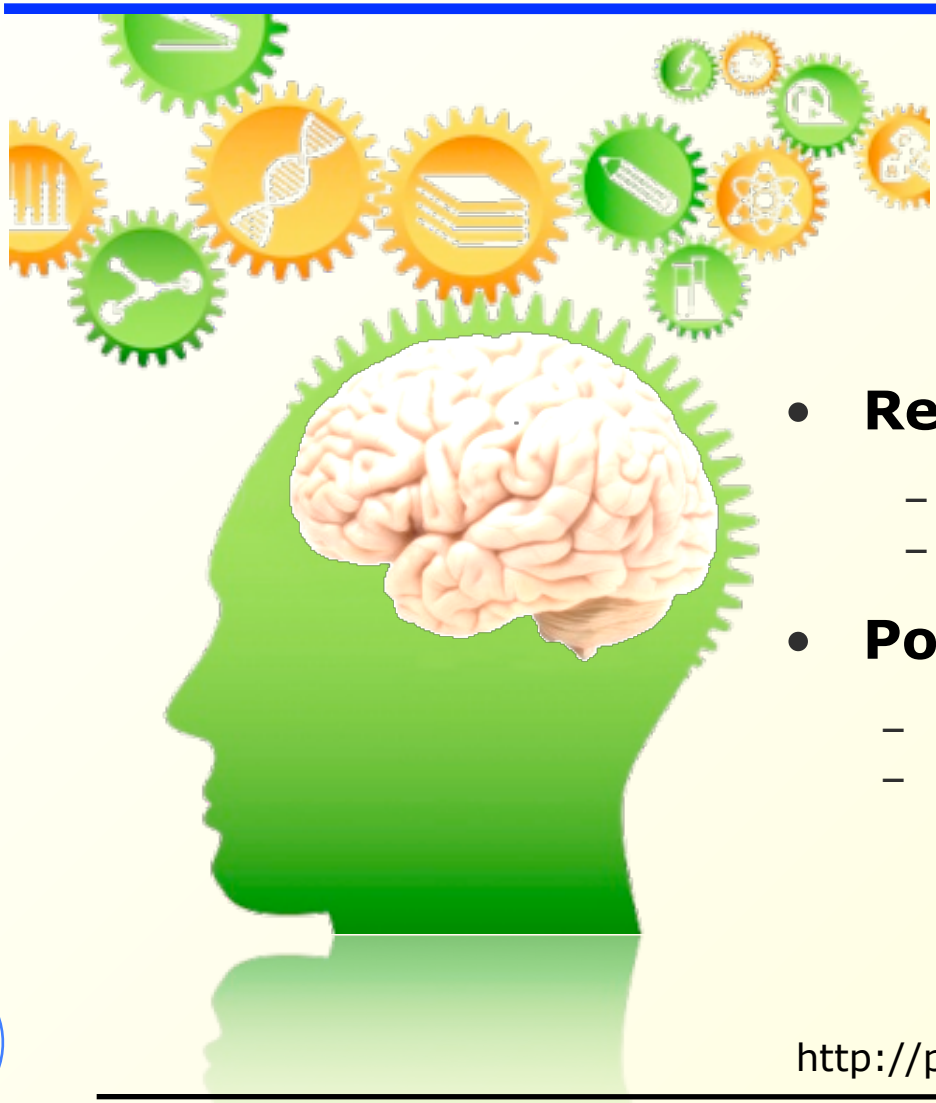
Systèmes à Base de Connaissances



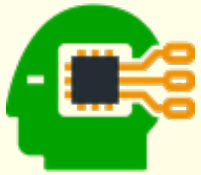
Le système CLIPS

- **Représentation des connaissances**
  - Aspects déclaratifs : les faits et les règles
  - Aspects fonctionnels et impératifs
- **Portage et encapsulation**
  - Java Expert System Shell
  - PyCLIPS Python Wrapper

http://perso.univ-lemans.fr/~jlehuen/clips/

Ce document n'est pas un manuel de référence mais un support de cours
Les documentations de CLIPS sont téléchargeables sur le site web suivant :
http://www.clipsrules.net/Documentation.html

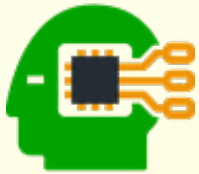
Une autre façon de programmer...



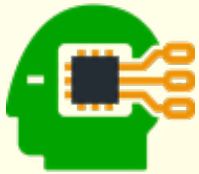
- **Comment aborder les problèmes suivants :**
  (d'un point de vue informatique)

  - *Cette voiture ne démarre pas, que dois-je faire ?*
  - *Comment interpréter les symptômes de ce patient ?*
  - *Est-ce le moment de vendre mes actions EuroTunnel ?*
  - *Une OPA contre la Société Générale peut-elle réussir ?*
  - *Dois-je protéger ma tour ou bien prendre son cavalier ?*
  - *À quelle heure dois-je me lever pour être en cours à 8h ?*
  - *Comment caser ces bagages dans le coffre de ma voiture ?*
  - *Quelle sémantique associer à cette chaîne de caractères ?*
  - *Comment répondre à cette question ?*

Une autre façon de programmer...

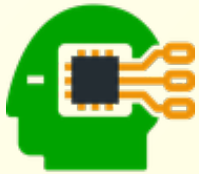


- **Selon la nature du problème, on a plusieurs méthodes :**
  - Formalisations de type configurations / contraintes
  - Exemples : cryptarithmétique, huit reines, grille de sudoku, etc.
  - Formalisations de type buts / opérateurs de décomposition
  - Exemples : preuve / résolution de problèmes, planification, etc.
  - Formalisations de type états / opérateurs de changement d'état
  - Exemples : cruches, loup-chèvre-choux, jeux de stratégies, etc.
  - Formalisations de type arbre de décision / règles heuristiques
  - Exemples : diagnostique médical ou de pannes, prise de décision, etc.
  - *Et ce ne sont que les méthodes symboliques !*



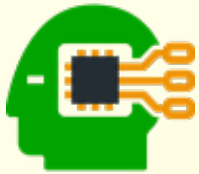
Une autre façon de programmer...

- **Différents paradigmes de programmation :**
  - Programmation impérative (Assembleur, Fortran, Pascal, C, etc.)
    - Le programme est constitué d'une suite d'instructions
    - La solution est le résultat de changements d'état de variables
    - Adapté aux architectures des ordinateurs (modèle de Von Neumann)
  - Programmation fonctionnelle (Lisp, Scheme, Caml, Haskell, etc.)
    - Le programme est constitué d'un ensemble de fonctions
    - La solution est le résultat d'un enchaînement d'appels de fonctions
    - Adapté aux calculs mathématiques ou formels (lambda-calcul)
    - Avantage : contourne les problèmes liés à la concurrence et aux effets de bord
  - Programmation orientée objet (Smalltalk, Ada, Java, Ruby, etc.)
    - Le programme est constitué d'un ensemble structurés d'objets
    - La solution est le résultat d'une suite d'interactions entre objets
    - Adapté au Génie Logiciel et aux gros projets industriels



Une autre façon de programmer...

- **Différents paradigmes de programmation (suite) :**
  - Programmation inférentielle (Prolog, Datalog, CLIPS, Jess, etc.)
    - Le programme est décrit à l'aide de règles de raisonnement (inférences)
    - Adapté aux problèmes combinatoires et à l'Intelligence Artificielle
  - Programmation logique et par contrainte (Prolog, Datalog, ECLiPSe)
    - Le programme est constitué d'un ensemble de clauses de Horn
    - La solution est le résultat du parcours d'un arbre de recherche
    - Fondé sur la logique des prédicats et la résolution de Robinson
  - Programmation à base de connaissances (CLIPS, OPS, Jess)
    - Le programme est constitué d'une base de règles de production
    - La solution est le résultat de la saturation d'une base de faits
    - Formalisme moins rigide (plus naturel) que celui de Prolog



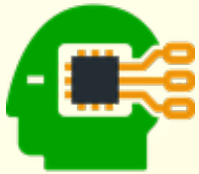
Une autre façon de programmer...

- **Quelques bonnes raisons pour envisager de développer un système à base de connaissances (SBC) :**
  - On ne dispose pas d'une vision globale du problème
  - On ne dispose pas d'une méthode algorithmique simple
  - Les données du problème sont incomplètes ou évolutives
  - Il n'y a pas forcément qu'une seule solution au problème
  - On peut être amené à tâtonner pour trouver une solution
  - On peut être amené à choisir entre plusieurs directions
  - Le problème se décrit bien à l'aide d'un ensemble de règles
  - On voudrait s'inspirer d'une méthode de résolution humaine

On ne dispose pas d'un corpus d'exemples suffisamment conséquent permettant d'envisager un apprentissage-machine

=> On peut toujours tout faire avec des langages "classiques" mais on risque de perdre du temps ou de réinventer la roue !

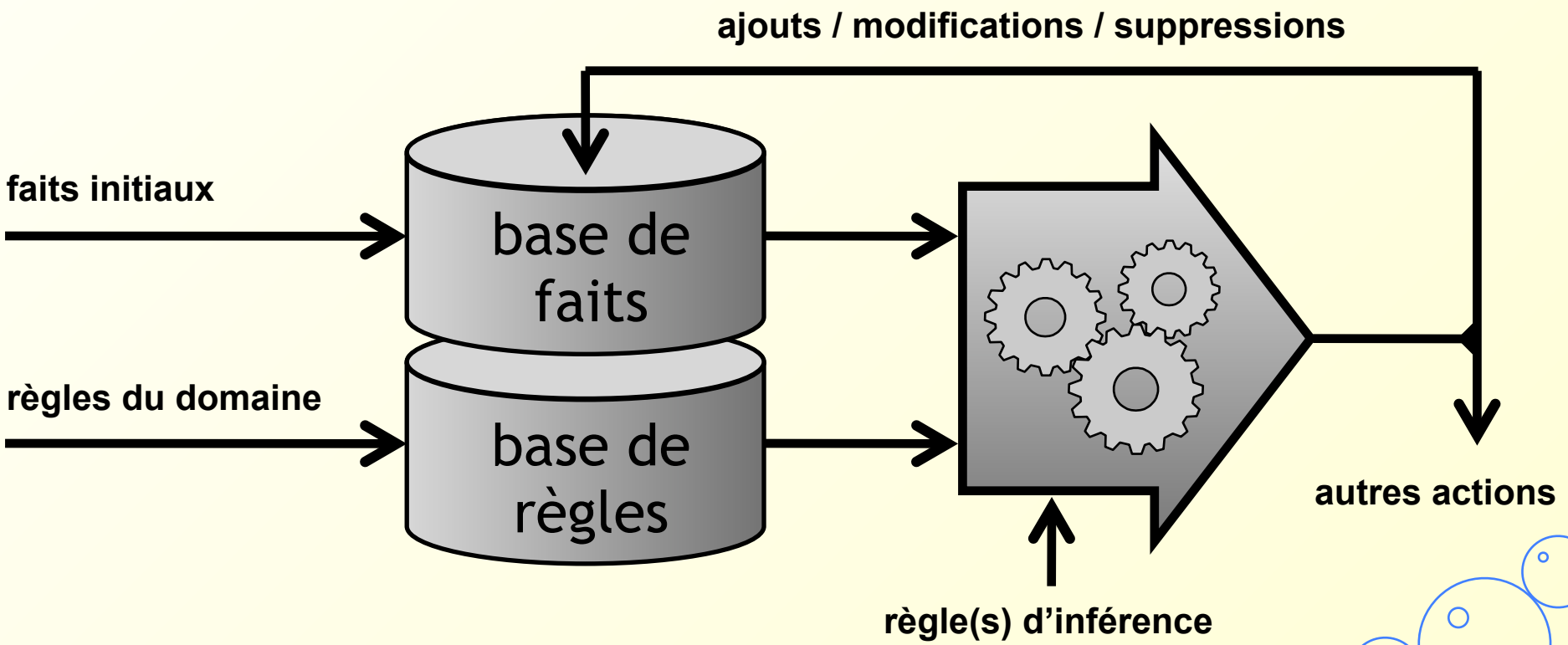
Connaissances d'un SBC



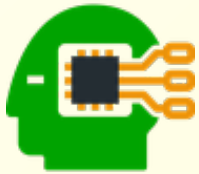
- **Partie déclarative :**
  - Fait : connaissance déclarative (information) sur le monde réel
  - Règle : connaissance déclarative sur la manipulation des faits
    - En logique monotone : ajouts de faits uniquement
    - En logique non-monotone : ajouts / modifications / retraits de faits
- **Partie algorithmique :**
  - Moteur d'inférences : composant logiciel qui effectue des raisonnements sur les connaissances dont il dispose :
    - Manipule les faits (ajouts / modifications / retraits) en utilisant les règles
    - Fondé sur une ou plusieurs règles d'inférences (ex : déduction, induction, *etc.*)
    - La trace du raisonnement peut être consultée :
      - Durant les phases de conception ou de débugage
      - Pour produire des explications sur la résolution

Architecture d'un SBC

- La base de faits contient les faits initiaux, puis les faits déduits
- La base de règles contient les règles qui exploitent la base de faits
- Le moteur d'inférences applique les règles aux faits



C'est quoi une inférence ?



Modus Ponens

Modus Tollens

Déductions sans variable :

(homme Socrate) est vrai
(homme Socrate) ➞ (mortel Socrate)

(mortel Socrate) est vrai

(mortel McLeod) est faux
(homme McLeod) ➞ (mortel McLeod)

(homme McLeod) est faux

Déductions avec variables :

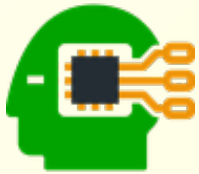
(homme Socrate) est vrai
(homme ?x) ➞ (mortel ?x)

Subst : { ?x = Socrate }
(mortel Socrate) est vrai

(mortel McLeod) est faux
(homme ?x) ➞ (mortel ?x)

Subst : { ?x = McLeod }
(homme McLeod) est faux

Caractéristiques de CLIPS 6

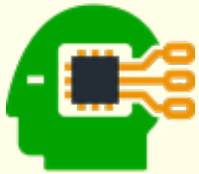


- **CLIPS = C-Language Integrated Production System :**
  - Initialement développé par le département *Artificial Intelligence* du *Lyndon B. Johnson Space Center* (NASA) dans les années 80
  - Intègre 4 paradigmes : inférentiel + fonctionnel + impératif + objet
  - Moteur d'inférences d'ordre 1 fonctionnant en chaînage avant
  - Implémentation en langage C de l'algorithme RETE (efficacité)
  - Un mode d'exécution en ligne de commande + fichiers batch
  - Des environnements de développement (sous MacOSX et Windows)
  - Code libre et open-source (code C ANSI lisible et documenté)
  - Porté vers d'autres langages (C++, Java, PHP, Python, etc.)
  - Logiciel stable et efficace (v6.40 disponible depuis avril 2021)

  => CLIPS peut être utilisé comme une librairie C/C++

  => Jess (Java Expert System Shell) est un portage Java

  => PyCLIPS est un wrapper Python pour accéder à CLIPS



CLIPS ou Prolog ?

- **Différences entre Prolog et CLIPS :**
  - Prolog est fondé sur une méthode de résolution en logique des prédicats (méthode de Robinson) qui revient à construire un arbre ET/OU à partir d'un but à atteindre afin de rechercher des prédicats qui permettraient d'atteindre ce but => chaînage arrière
  - CLIPS est un fondé sur un principe d'inférence qui consiste à enchaîner des déductions à partir des données à l'aide de règles de production et à rechercher le ou les faits qui constitueraient une solution à un problème ou à une décision à prendre => chaînage avant
  - Une résolution Prolog s'apparente à une preuve mathématique
  - Une résolution CLIPS s'apparente à un raisonnement humain
  - *Chacun des deux langages a ses avantages et ses inconvénients*

Environnement de développement

timlechat.clp

Dialog

```
(deffacts faits-initiaux
    (propriete Tim ronronne)
    (propriete Tim est-familier)
    (propriete Tim vit-dans-la-maison)
    (pere-de Felix Tim))

(defrule regle-1
    (propriete ?x ronronne)
    =>
    (assert (propriete ?x est-un-chat)))

(defrule regle-2
    (propriete ?x est-un-chat)
    =>
    (assert (propriete ?x est-un-mamifere))
    (assert (propriete ?x possede-des-griffes)))

(defrule regle-3
    (propriete ?x ronronne)
    (propriete ?x est-familier)
    (propriete ?x vit-dans-la-maison)
    =>
    (assert (propriete ?x est-un-animal-domestique))
    (assert (propriete ?x est-dorlote)))
```

Editeur

Dialog

Dir: ~/Documents/Universite/Enseignement/M1-CLIPS/Cours

Pause

```
         CLIPS (6.30 8/6/13)
CLIPS> (load timlechat.clp)
Defining deffacts: faits-initiaux
Defining defrule: regle-1 +j+j
Defining defrule: regle-2 +j+j
Defining defrule: regle-3 =j+j+j+j
Defining defrule: regle-4 +j+j+j
TRUE
CLIPS> (reset)
CLIPS>
```

Console en mode texte

Agenda

Dialog | Reset | Run | Step | Halt

Focus Stack
MAIN

Salience	Rule	Basis
0	regle-4	f-4,f-3
0	regle-4	f-4,f-2
0	regle-4	f-4,f-1
0	regle-3	f-1,f-2,f-3
0	regle-1	f-1

Agenda

Facts

Dialog | Display Defaulted Values | search

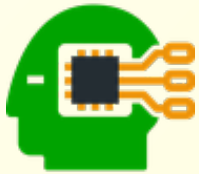
Module
MAIN

Index	Template
0	initial-fact
1	propriete
2	propriete
3	propriete
4	pere-de

Slot	Value
implied	(Tim vit-dans-la-maison)

Base de faits

Premier programme CLIPS



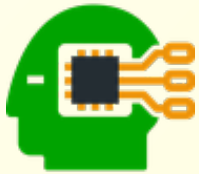
- **Soit le programme suivant (timlechat_v1.clp) :**

```
(deffacts faits-initiaux

      (ronronne Tim)
      (est-familier Tim)
      (vit-dans-la-maison Tim))

(defrule regle-1 "Qui ronronne est un chat"

      (ronronne ?x)
      =>
      (assert (est-un-chat ?x)))

(defrule regle-2 "Un chat est un mammifère qui possède des griffes"

      (est-un-chat ?x)
      =>
      (assert (est-un-mammifere ?x))
      (assert (possede-des-griffes ?x)))

(defrule regle-3 "Qui ronronne ET est familier ET vit dans la maison
                  est un animal domestique ET est dorloté"

      (ronronne ?x)
      (est-familier ?x)
      (vit-dans-la-maison ?x)
      =>
      (assert (est-un-animal-domestique ?x))
      (assert (est-dorlote ?x)))
```

"Qui ronronne est un chat" n'est pas équivalent à : *"Un chat ronronne"*

La variable ?x impose une contrainte entre les trois préconditions

Premier programme CLIPS



- **Note concernant la syntaxe et les conventions d'écriture :**
  - La syntaxe est de type parenthésée préfixée (inspirée du langage Lisp)
    - *Toute structure, commande ou appel de fonction commence par une parenthèse ouvrante et se termine par une parenthèse fermante !*
    - *Les paramètres des fonctions ne sont pas délimités par des parenthèses !*

    - Pas d'espace après une parenthèse ouvrante ou avant une parenthèse fermante
    - Un espace avant une parenthèse ouvrante et après une parenthèse fermante
    - Pas d'espace entre deux parenthèse ouvrante ou deux parenthèses fermantes
    - C'est plus lisible de fermer les parenthèses en fin de ligne
    - En revanche, bien respecter les indentations à la ligne pour les listes non fermées

    - Toutes les variables commencent par un point d'interrogation
    - Les commentaires sont monolignes et commencent par un point-virgule
    - Les tokens sont codés en UTF-8 depuis la version 6.30 mais on évite d'utiliser des caractères accentués car cela peut générer des erreurs difficiles à déceler

Premier programme CLIPS

- **Chargement de la base de connaissances :**

```
CLIPS> (load timlechat_v1.clp)
Defining deffacts: faits-initiaux
Defining defrule: regle-1 +j+j
Defining defrule: regle-2 +j+j
Defining defrule: regle-3 =j+j+j+j
TRUE
CLIPS>
```

- **Initialisation de l'agenda et affichage de la base de faits :**

```
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (watch activations)
CLIPS> (reset)
==> f-1     (ronronne Tim)
==> Activation 0      regle-1: f-1
==> f-2     (est-familier Tim)
==> f-3     (vit-dans-la-maison Tim)
==> Activation 0      regle-3: f-1,f-2,f-3
CLIPS>
```

```
CLIPS> (facts)
f-1     (ronronne Tim)
f-2     (est-familier Tim)
f-3     (vit-dans-la-maison Tim)
For a total of 3 facts.
CLIPS>
```

- La commande (reset) place les faits initiaux dans la BDF et initialise l'exécution du programme => Activations

Premier programme CLIPS

- **Exécution du programme (avec affichage de la trace) :**

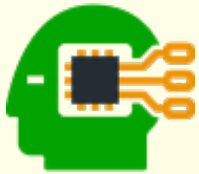
```
CLIPS> (run)
FIRE    1 regle-3: f-1,f-2,f-3
==> f-4     (est-un-animal-domestique Tim)
==> f-5     (est-dorlote Tim)
FIRE    2 regle-1: f-1
==> f-6     (est-un-chat Tim)
==> Activation 0      regle-2: f-6
FIRE    3 regle-2: f-6
==> f-7     (est-un-mammifere Tim)
==> f-8     (possede-des-griffes Tim)
CLIPS>
```

- **Affichage de la base de faits avant et après l'exécution :**

```
CLIPS> (facts)
f-1     (ronronne Tim)
f-2     (est-familier Tim)
f-3     (vit-dans-la-maison Tim)
For a total of 3 facts.
CLIPS>
```

```
CLIPS> (facts)
f-1     (ronronne Tim)
f-2     (est-familier Tim)
f-3     (vit-dans-la-maison Tim)
f-4     (est-un-animal-domestique Tim)
f-5     (est-dorlote Tim)
f-6     (est-un-chat Tim)
f-7     (est-un-mammifere Tim)
f-8     (possede-des-griffes Tim)
For a total of 8 facts.
CLIPS>
```

Les faits ordonnés



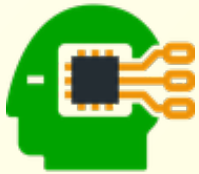
- **Définition d'un fait ordonné :**
  - Un fait ordonné est une liste plate non typée composé d'une relation symbolique suivi d'une séquence (éventuellement vide) de termes
  - La sémantique d'un fait n'est pas implicite, il peut être utile de la préciser / l'expliciter en ajoutant un commentaire :

```
(lock)                                  ; Verrou activé
(age-de Pierre 18)                      ; Pierre a 18 ans
(pere-de Pierre Paul)                   ; Paul est le père de Pierre
(est-dans-sa-baignoire Pierre)          ; Pierre est dans sa baignoire
(adresse-de Pierre "41 rue des Fauvettes")  ; Pierre habite au 41 rue des Fauvettes
(phrase le chat mange la souris)        ; Le chat mange la souris
```

  - La relation doit obligatoirement être un symbole (token)
  - Les termes peuvent être des symboles, des entiers, des floats, des chaînes de caractères, ou encore des références vers d'autres faits

  - *Attention : il n'y a aucun contrôle sur le type des termes !*

Les règles de production



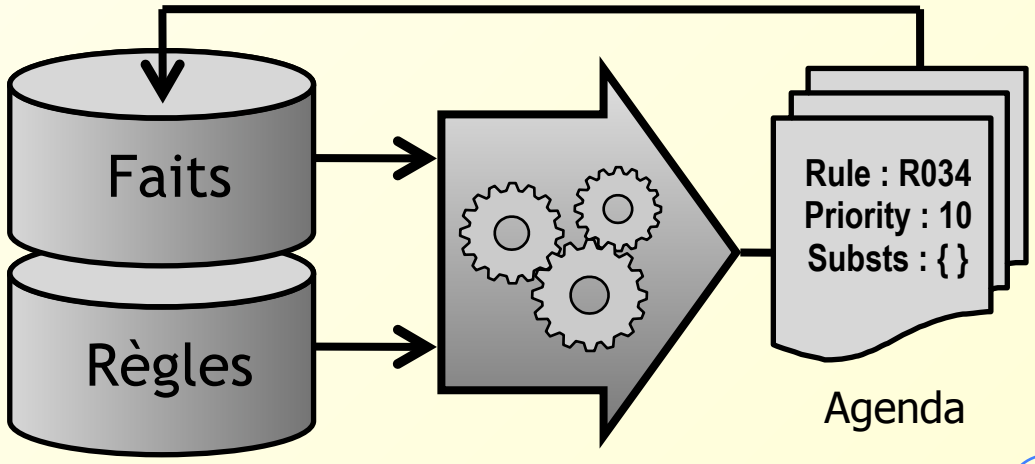
- **Définition d'une règle de production :**
  - Une règle est constituée d'une série de préconditions suivie d'une série d'actions à exécuter si toutes les préconditions sont vérifiées
    - L'ensemble des préconditions est appelé membre gauche ou LHS (Left Hand Side)
    - L'ensemble des actions est appelé membre droit ou RHS (Right Hand Side)
  - Un filtre (pattern) est une précondition qui permet de tester l'existence de faits selon un mécanisme de *pattern-matching*
    - Un filtre est une liste de termes qui ressemble structurellement aux faits
    - Un filtre contient une relation puis des constantes littérales et/ou des variables
    - Les constantes littérales et les variables liées imposent des contraintes
  - La commande (assert …) dans un RHS permet d'ajouter (produire) un nouveau fait dans la base de faits
    - Il est ainsi possible d'enchaîner des déductions tout en conservant chaque nouvel élément déduit, même si le processus de déduction n'est pas terminé

Le moteur d'inférence

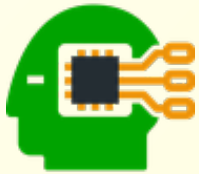
- **Quelques définitions :**
  - Une règle déclenchable est une règle dont toutes les conditions du membre gauche (préconditions) sont vérifiées
  - Une activation est l'association d'une règle déclenchable, des faits et des substitutions qui permettent de la déclencher
  - Un agenda est une pile d'activations (règles déclenchables en attente)

- **Algorithme simplifié :**

- Initialiser l'agenda => premières activations
- Tant qu'il y a des activations dans l'agenda
  - Déclencher la première règle
  - Mettre à jour l'agenda :
    - Empiler les nouvelles activations
    - Supprimer les activations obsolètes



Tim le chat - version 2



- **Nouveau problème :**
  - Un fils doit hériter de toutes les propriétés de son père :

```
(deffacts faits-initiaux
     (ronronne Tim)
     (est-familier Tim)
     (vit-dans-la-maison Tim)
     (pere-de Felix Tim))

(defrule regle-1
     (ronronne ?x)
     =>
     (assert (est-un-chat ?x)))

(defrule regle-2
     (est-un-chat ?x)
     =>
     (assert (est-un-mammifere ?x))
     (assert (possede-des-griffes ?x)))

(defrule regle-3
     (ronronne ?x)
     (est-familier ?x)
     (vit-dans-la-maison ?x)
     =>
     (assert (est-un-animal-domestique ?x))
     (assert (est-dorlote ?x)))
```

```
(defrule regle-4-ronronne
     (pere-de ?fils ?pere)
     (ronronne ?pere)
     =>
     (assert (ronronne ?fils)))

(defrule regle-4-est-familier
     (pere-de ?fils ?pere)
     (est-familier ?pere)
     =>
     (assert (est-familier ?fils)))

(defrule regle-4-vit-dans-la-maison
     (pere-de ?fils ?pere)
     (vit-dans-la-maison ?pere)
     =>
     (assert (vit-dans-la-maison ?fils)))
```

Une règle pour chaque propriété

Tim le chat - version 2

- **Nouveau problème (suite) :**
  - Un fils doit hériter de toutes les propriétés de son père :

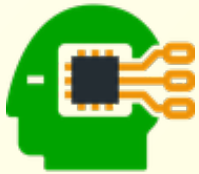
```
(defrule regle-4
      (pere-de ?fils ?pere)
      (?prop ?pere)
      =>
      (assert (?prop ?fils)))
```

Une règle unique pour toutes les propriétés ?

```
(defrule regle-4-ronronne
      (pere-de ?fils ?pere)
      (ronronne ?pere)
      =>
      (assert (ronronne ?fils)))

(defrule regle-4-est-familier
      (pere-de ?fils ?pere)
      (est-familier ?pere)
      =>
      (assert (est-familier ?fils)))

(defrule regle-4-vit-dans-la-maison
      (pere-de ?fils ?pere)
      (vit-dans-la-maison ?pere)
      =>
      (assert (vit-dans-la-maison ?fils)))
```

Attention : interdiction de placer une variable au niveau de la relation (car moteur d'inférence d'ordre 1)

Tim le chat - version 2



- **Solution pour raisonner sur les relations :**
  - Transformer les relation en donnée et insérer une nouvelle relation :

```
(deffacts faits-initiaux
     (propriete ronronne Tim)
     (propriete est-familier Tim)
     (propriete vit-dans-la-maison Tim)
     (pere-de Felix Tim))

(defrule regle-1
     (propriete ronronne ?x)
     =>
     (assert (propriete est-un-chat ?x)))

(defrule regle-2
     (propriete est-un-chat ?x)
     =>
     (assert (propriete est-un-mammifere ?x))
     (assert (propriete possede-des-griffes ?x)))
```

Les propriétés ne sont plus les relations

```
(defrule regle-3
     (propriete ronronne ?x)
     (propriete est-familier ?x)
     (propriete vit-dans-la-maison ?x)
     =>
     (assert (propriete est-un-animal-domestique ?x))
     (assert (propriete est-dorlote ?x)))

(defrule regle-4
     (pere-de ?fils ?pere)
     (propriete ?prop ?pere)
     =>
     (assert (propriete ?prop ?fils)))
```

Elles peuvent donc être manipulées

Tim le chat - version 2

- **Exécution du programme (avec affichage de la trace) :**

```
CLIPS> (run)
FIRE   1 regle-4: f-4,f-3
==> f-5  (propriete vit-dans-la-maison Felix)
FIRE   2 regle-4: f-4,f-2
==> f-6  (propriete est-familier Felix)
FIRE   3 regle-4: f-4,f-1
==> f-7  (propriete ronronne Felix)
==> Activation 0      regle-1: f-7
==> Activation 0      regle-3: f-7,f-6,f-5
FIRE   4 regle-3: f-7,f-6,f-5
==> f-8  (propriete est-un-animal-domestique Felix)
==> f-9  (propriete est-dorlote Felix)
FIRE   5 regle-1: f-7
==> f-10 (propriete est-un-chat Felix)
==> Activation 0      regle-2: f-10
FIRE   6 regle-2: f-10
==> f-11 (propriete est-un-mammifere Felix)
==> f-12 (propriete possede-des-griffes Felix)
FIRE   7 regle-3: f-1,f-2,f-3
==> f-13 (propriete est-un-animal-domestique Tim)
==> Activation 0      regle-4: f-4,f-13
==> f-14 (propriete est-dorlote Tim)
==> Activation 0      regle-4: f-4,f-14
FIRE   8 regle-4: f-4,f-14
FIRE   9 regle-4: f-4,f-13
FIRE  10 regle-1: f-1
==> f-15 (propriete est-un-chat Tim)
==> Activation 0      regle-4: f-4,f-15
==> Activation 0      regle-2: f-15
FIRE  11 regle-2: f-15
==> f-16 (propriete est-un-mammifere Tim)
==> Activation 0      regle-4: f-4,f-16
==> f-17 (propriete possede-des-griffes Tim)
==> Activation 0      regle-4: f-4,f-17
FIRE  12 regle-4: f-4,f-17
FIRE  13 regle-4: f-4,f-16
FIRE  14 regle-4: f-4,f-15
CLIPS>
```

- Rouge = déclenchement des règles
- Bleu = évolution de la base de fait (assertions)
- Vert = évolution de l'agenda (activations)

Tim le chat - version 2

- **Affichage de la base de faits après exécution :**

```
CLIPS> (facts)
f-1     (propriete ronronne Tim)
f-2     (propriete est-familier Tim)
f-3     (propriete vit-dans-la-maison Tim)
f-4     (pere-de Felix Tim)
f-5     (propriete vit-dans-la-maison Felix)
f-6     (propriete est-familier Felix)
f-7     (propriete ronronne Felix)
f-8     (propriete est-un-animal-domestique Felix)
f-9     (propriete est-dorlote Felix)
f-10    (propriete est-un-chat Felix)
f-11    (propriete est-un-mammifere Felix)
f-12    (propriete possede-des-griffes Felix)
f-13    (propriete est-un-animal-domestique Tim)
f-14    (propriete est-dorlote Tim)
f-15    (propriete est-un-chat Tim)
f-16    (propriete est-un-mammifere Tim)
f-17    (propriete possede-des-griffes Tim)
For a total of 17 facts.
CLIPS>
```

- Le programme s'est arrêté lorsque plus aucune règle ne pouvait être déclenchée (agenda vide) => la base de faits est dite saturée
- Des 4 faits initiaux, on est passé à 17 faits

Utilisation des variables

- **Variable libre *vs*. variable liée :**
  - Les variables permettent de propager des contraintes entre les filtres
    - Une variable libre dans un filtre peut prendre n'importe quelle valeur
    - Une variable liée dans un filtre permet d'imposer une valeur sur un terme
    - Les variables peuvent également être réutilisées dans les RHS des règles :

```
(defrule regle-4
     (pere-de ?fils ?pere)
     (propriete ?prop ?pere)
     =>
     (assert (propriete ?prop ?fils)))
```

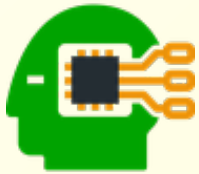
ici les variables ?fils et ?pere sont libres

ici la variable ?pere est liée => contrainte
et la variable ?prop est libre

ici les variables ?fils et ?prop
sont forcément liées

  - Attention : il n'est pas possible de lier des variables dans un LHS autrement qu'au travers du mécanisme de *pattern matching !*
  - Il est également possible d'utiliser le joker ? dans un filtre

Les fait structurés



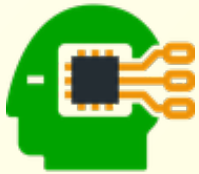
- **Des structures, comme en langage C :**
  - Structures de données apparentées au "struct" du langage C contenant des attributs (*slots*) identifiés par un nom :

```
(employe (prenom "Jacques")
         (nom "Dupont")
         (age 27)
         (contrat CDD)
         (departements 72 53 61))
```

  - Les faits structurés doivent être préalablement déclarés :

```
(deftemplate employe "Les employés de mon entreprise"

     (slot nom                (type STRING) (default ?NONE))
     (slot prenom             (type STRING) (default ?DERIVE))
     (slot age                (type INTEGER) (range 18 ?VARIABLE))
     (slot contrat            (allowed-values CDD CDI) (default CDD))
     (multislot departements  (type INTEGER)))
```

Types et autres contraintes sur
la valeur des attributs (optionnel)



Les fait structurés

- **Les contraintes de type et de valeur :**

```
(type <type>+)

    <type> ::= SYMBOL|INTEGER|FLOAT|STRING|FACT-ADDRESS|...

(allowed-symbols <symbol>+)      Pour les valeurs de type SYMBOL
(allowed-integers <integer>+)    Pour les valeurs de type INTEGER
(allowed-values <value>+)        Pour tout type de valeurs
```

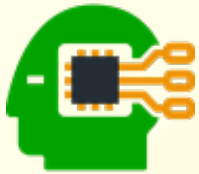
- **Les contraintes d'intervalle et de cardinalité :**

```
(range <range-spec> <range-spec>)

     <range-spec> ::= <integer>|<float>|?VARIABLE

(cardinality <card-spec> <card-spec>)

     <card-spec> ::= <integer>|?VARIABLE
```

- **Les valeurs par défaut :**

```
(default <value>|?NONE|?DERIVE)
```

Suppression d'un fait



- **Il faut mémoriser une référence dans une variable :**

```
(defrule supprime-prop "Suppression d'un fait propriété"
     (deces-de ?x)
     ?p <- (propriete ?x ?)
     =>
     (retract ?p))
```

La variable ?p sera de type FACT-ADDRESS

- Les variables de type FACT-ADDRESS permettent de manipuler les faits
- Lorsqu'un fait est supprimé, cela peut entrainer des modifications au sein de l'agenda => les activations qui dépendent du fait sont retirées

- Il est possible de supprimer plusieurs faits avec un seul retract :

```
(retract ?f1 ?f2 ?f3)
```

  - Pour éviter les déductions inutiles, les règles de suppression ont souvent intérêt à être prioritaires sur les autres règles => notion de salience (cf. plus loin)

Modification d'un fait

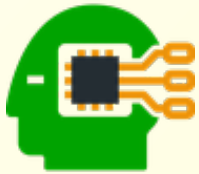
- **Cas des faits ordonnés :**

```
(deffacts faits-initiaux
     (pompe 1 statut inactive)
     (pompe 2 statut inactive))

(defrule activer-pompes
     (activer-pompes)                          ; Un fait déclencheur
     ?p <- (pompe ?num statut inactive)        ; Un fait à modifier
     =>
     (retract ?p)                              ; Suppression du fait à modifier
     (assert (pompe ?num statut active)))      ; Assertion du fait à modifier
```

- **Cas des faits structurés :**

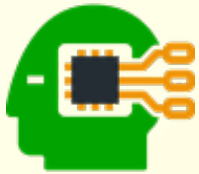
```
(deftemplate pompe
     (slot numero (type INTEGER) (default ?NONE))
     (slot statut (allowed-values active inactive) (default inactive)))

(deffacts faits-initiaux
     (pompe (numero 1))
     (pompe (numero 2)))

(defrule activer-pompes
     (activer-pompes)                    ; Un fait déclencheur
     ?p <- (pompe (statut inactive))     ; Un fait à modifier
     =>
     (modify ?p (statut active)))        ; Modification d'un slot du fait
```

Faisons un peu le point…



- **Principaux constructeurs :**
  - defrule : définir une règle de production
  - deffacts : définir un ensemble de faits initiaux
  - deftemplate : définir un type de faits structurés
- **Principales commandes :**
  - clear : réinitialiser complètement l'environnement
  - load : charger un fichier de constructeurs
  - reset : initialiser le moteur d'inférence => base de faits, agenda, etc.
  - run : lancer le moteur d'inférences jusqu'à saturation de la BDF
- **Modification de la base de faits :**
  - assert : ajouter un nouveau fait (retourne FALSE si le fait existe déjà)
  - retract : retirer un fait existant à partir de son fact-address
  - modify : modifier un fait structuré à partir de son fact-address

Vérifier l'absence d'un fait



- **Revient à tester la non-présence du fait dans la base de faits :**

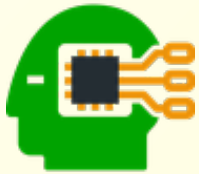
```
(deffacts faits-initiaux

     (personne Pierre)
     (personne Paul)

     (recherche Pierre)
     (recherche Jacques))
```

```
(defrule presence

     (recherche ?prenom)
     (personne ?prenom)
     =>
     (printout t ?prenom " est présent" crlf))

(defrule absence

     (recherche ?prenom)
     (not (personne ?prenom))
     =>
     (printout t ?prenom " est absent" crlf))
```

  - La précondition (not <pattern>) est vérifiée lorsqu'il n'y a aucun fait dans la base de faits qui correspond au filtre <pattern>

- **Fonctions d'affichage et de saisie :**

  - Affichage : `(printout t [<chaine>|<variable>|crlf]+ )`
  - Saisie : `(readline)` retourne une chaîne de caractères



Les variables multivaluées

- **Les faits ordonnés sont par définition des listes :**
  - Il est donc possible de manipuler des listes de valeurs :
    - En utilisant des fonctions appropriées (insertion, suppression, extraction, etc.)
    - En utilisant des variables multivaluées dans les filtres des LHS des règles
  - Les variables multivaluées sont liées avec des listes de termes (éventuellement vides) au cours du processus de pattern-matching :

```
    Fait = (a b c d e)

Filtre 1 = (a $?x d ?)          ?x = (b c)

Filtre 2 = (a $?x)              ?x = (b c d e)
```

    - Attention : les filtres avec plusieurs multivalués génèrent de la combinatoire :

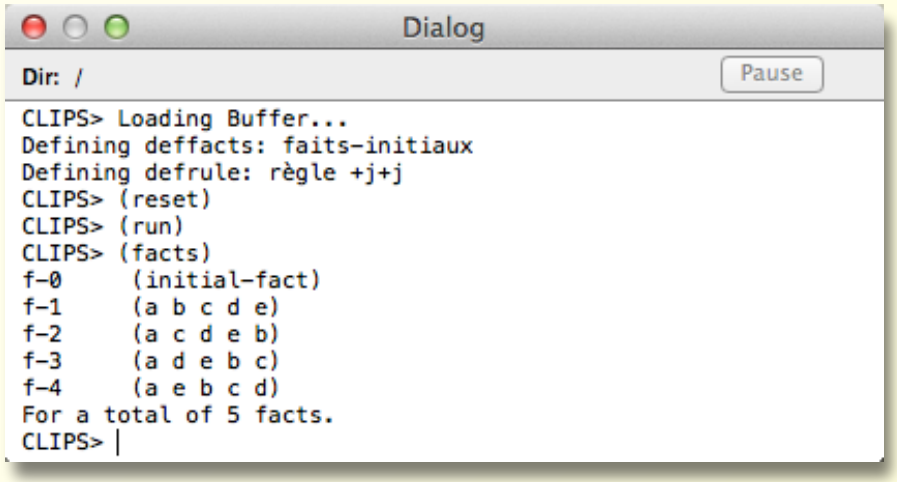
```
Filtre 3 = (a $?x $?y)          ?x = ()          ?y = (b c d e)
                                ?x = (b)         ?y = (c d e)
                                ?x = (b c)       ?y = (d e)
                                ?x = (b c d)     ?y = (e)
                                ?x = (b c d e)   ?y = ()
```

    - Il est également possible d'utiliser le joker $? dans un filtre

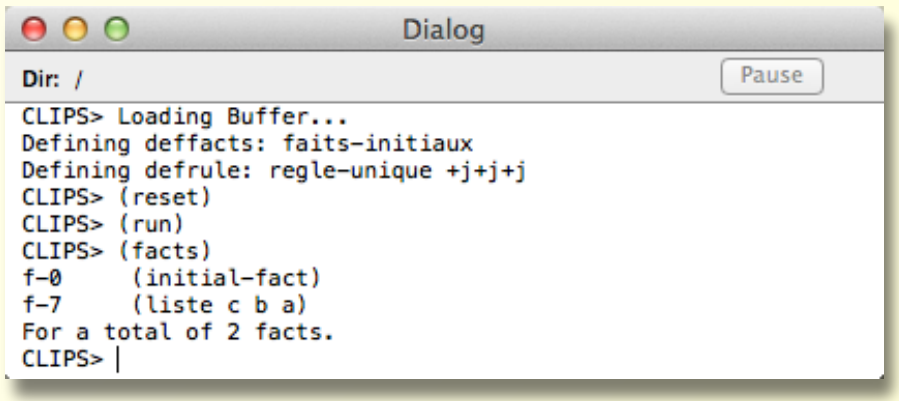
Les variables multivaluées

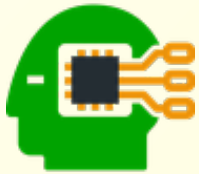
- **Que font les programmes suivants ?**

```
(deffacts faits-initiaux

     (a b c d e))

(defrule regle-unique

     (a $?x $?y)
     =>
     (assert (a ?y ?x)))
```




```
(deffacts faits-initiaux

     (liste)
     (data a)
     (data b)
     (data c))

(defrule regle-unique

     ?f1 <- (liste $?x)
     ?f2 <- (data ?y)
     =>
     (retract ?f1 ?f2)
     (assert (liste ?x ?y)))
```





Syntaxe des préconditions

- **Précondition booléenne (*test conditional element*) :**
  - Permet de placer des expressions booléennes dans les LHS :
    - Intégration de connecteurs logiques (conjonction, disjonction, négation)
    - Intégration d'appels de fonctions et de prédicats (fonctions booléennes)

```
(defrule regle-bidon
          ...
          (test (<function-call>))
          ...
          =>
          ...
```

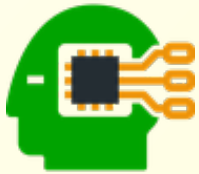
Prédicats de type : (numberp <expr>), (integerp <expr>), (floatp <expr>),
(stringp <expr>), (symbolp <expr>), (multifieldp <expr>)...

Comparaisons : (eq <expr> <expr>+), (neq <expr> <expr>+),
(= <expr> <expr>+), (> <expr> <expr>+), (>= <expr> <expr>+)...

Autres prédicats : (oddp <expr>), (evenp <expr>), (subsetp <expr> <expr>)...

Opérateurs booléens : (and <expr>+), (or <expr>+), (not <expr>)

Syntaxe des préconditions



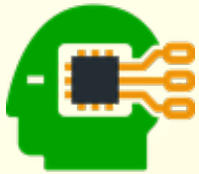
- **Combinaison de préconditions (*or/and conditional element*) :**
  - Permet de combiner des préconditions avec des OR et des AND.
  - Toutes les préconditions d'une règle sont reliées par un AND implicite.
  - La précondition OR permet d'écrire plusieurs règles en une seule :

```
(defrule regle-bidon-1
         ...
         (conditional element 1)
         ...
         =>
         action
```

```
(defrule regle-bidon-2
         ...
         (conditional element 2)
         ...
         =>
         action
```

```
(defrule regle-bidon
         ...
         (or (conditional element 1)
             (conditional element 2))
         ...
         =>
         action
```

Syntaxe des préconditions

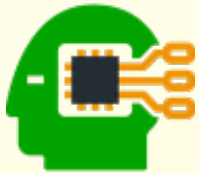


- **Vérifier l'absence d'un fait (*not conditional element*) :**
  - Permet de vérifier qu'aucun fait ne correspond à un pattern donné :

```
(defrule regle-bidon
         ...
         (not (conditional element))
         ...
         =>
         ...
```

  - *Inutile d'utiliser des variables dans une précondition NOT !*
  - En revanche, on peut utiliser des jokers monovalués ou multivalués.
  - A ne pas confondre avec le connecteur logique (not ...) :

```
(defrule regle-bidon
         ...
         (test (not <function-call>))
         ...
         =>
         ...
```



Syntaxe des préconditions

- **Vérifier la présence d'un fait (*exists conditional element*) :**
  - Permet de vérifier qu'il existe au moins 1 fait qui correspond à un pattern donné (permet d'éviter des déclenchements multiples) :

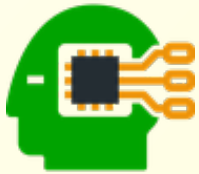
```
(defrule regle-bidon
        ...
        (exists (conditional element))
        ...
        =>
        ...
```

  - Exemples :

```
(deffacts faits-initiaux
     (relation a b c)
     (relation x y z))

(defrule regle-bidon
     (relation ? ? ?)
     =>
     (printout t "Ok" crlf))
```

```
(deffacts faits-initiaux
     (relation a b c)
     (relation x y z))

(defrule regle-bidon
     (exists (relation ? ? ?))
     =>
     (printout t "Ok" crlf))
```

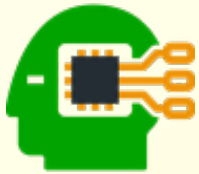
Syntaxe des préconditions



- **Contraintes intégrées (*connective constraints*) :**
  - Permet d'augmenter la puissance d'expression des filtres :
    - Intégration de connecteurs logiques (conjonction, disjonction, négation)
    - Intégration d'appels de fonctions booléennes

```
(personne (age 30))                          ; égale à une constante
(personne (age ?x))                          ; égale à une variable libre ou liée
(personne (age =(+ 30 ?x)))                  ; égale à une valeur calculée
(personne (age ~30))                         ; différente d'une constante
(personne (age ~?x))                         ; différente d'une variable liée
(personne (age ?x&~20))                      ; mémorisation et différente d'une constante
(personne (age ?x&~20&~30))                  ; combinaisons de tests
(personne (age ?x&:(oddp ?x)))               ; appel d'un prédicat (vrai/faux)...
(personne (age ?x&~:(oddp ?x)))                        ; plus une négation logique
(personne (age ?x&:(> ?x 30)&:(< ?x 40)))              ; ça se complique

(defrule regle-bidon
         (personne (nom ?x) (age ?y))
         (personne (nom ~?x) (age ?w&?y|=(* 2 ?y)))
         =>
         ...
```

Syntaxe des préconditions

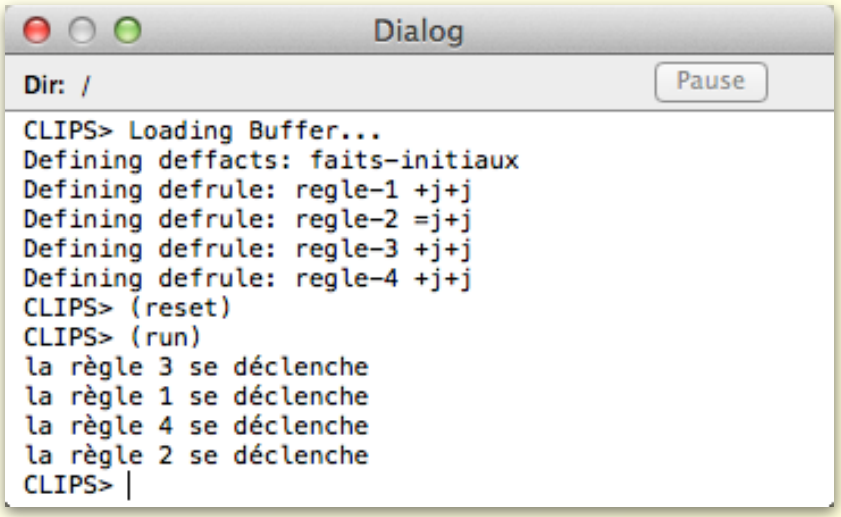


- **Synthèse :**
  - ***Pattern* Conditional Element :**
    - Syntaxe habituelle des filtres + possibilité de "contraintes intégrées"
  - ***Test* Conditional Element :**
    - Syntaxe : (test <function-call>) la fonction doit retourner TRUE ou FALSE
  - ***Or/And* Conditional Element :**
    - Syntaxes : (or <conditional-element>+) et (and <conditional-element>+)
    - Permet de combiner des préconditions de tout type
  - ***Not* Conditional Element :**
    - Syntaxe : (not <conditional-element>) possède deux sémantiques :
      - Si Pattern-CE : absence du fait décrit par le *pattern* (monde fermé)
      - Si tout autre type de condition : connecteur logique de négation
  - ***Exists* Conditional Element :**
    - Syntaxe : (exists <conditional-element>+) permet de ne générer qu'une activation, même si la règle peut être déclenchée de plusieurs façons

Déclenchement des règles

- **Dans quel ordre les règles sont-elles déclenchées ?**
  - L'ordre des déclenchements est déterminé par les règles suivantes :
    - On peut donner (avec modération) des priorités aux règles de -10000 à 10000
    - A priorités égales, toute règle nouvellement déclenchable est placée devant
    - A priorités égales, les premières règle écrites passent avant les autres

```
(deffacts faits-initiaux
      (toto)
      (titi))

(defrule regle-1
      (toto)
      =>
      (printout t "la règle 1 se déclenche" crlf)
      (assert (tutu)))

(defrule regle-2
      (toto)
      =>
      (printout t "la règle 2 se déclenche" crlf))

(defrule regle-3
      (titi)
      =>
      (printout t "la règle 3 se déclenche" crlf))

(defrule regle-4
      (tutu)
      =>
      (printout t "la règle 4 se déclenche" crlf))
```



Déclenchement des règles

- **Soit le programme suivant :**

```
(deftemplate pompe
      (slot numero (type INTEGER))
      (slot statut (allowed-values active inactive)
                   (default inactive)))

(deffacts faits-initiaux
      (pompe (numero 1))
      (pompe (numero 2))
      (activer-pompes))

(defrule activer-pompes
      (activer-pompes)
      ?p <- (pompe)
      =>
      (modify ?p (statut active)))
```

Que pensez-vous de ce programme ?

Attention : une règle peut se comporter comme une boucle infinie ou une fonction récursive sans condition d'arrêt !

- Lorsque vous avez un assert un modify ou un duplicate dans le RHS d'une règle, assurez-vous bien que vous n'avez pas créé les conditions d'un bouclage infini du moteur d'inférences !

- Dans ce cas, il ne faut modifier **que les pompes inactives :**

```
?p <- (pompe (statut inactive))
```

Le Loup, la chèvre et le choux

- **Problème :**
  - *Un fermier doit transborder un loup, une chèvre et un choux de la rive gauche à la rive droite d'un fleuve. À chaque traversée, il peut prendre au maximum un seul des trois protagonistes. En l'absence du fermier, le loup mange la chèvre, et la chèvre le choux !*

- **Résolution :**
  - Développer un graphe de recherche où chaque noeud contient des informations sur l'emplacement des protagonistes.
  - Une solution est donnée par un chemin entre la situation initiale et la situation finale.

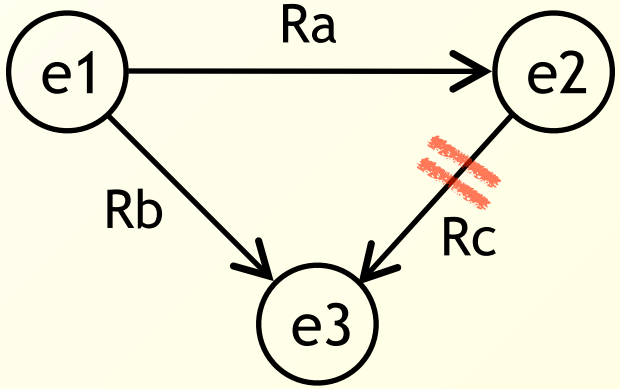
Quelle méthode de résolution ?

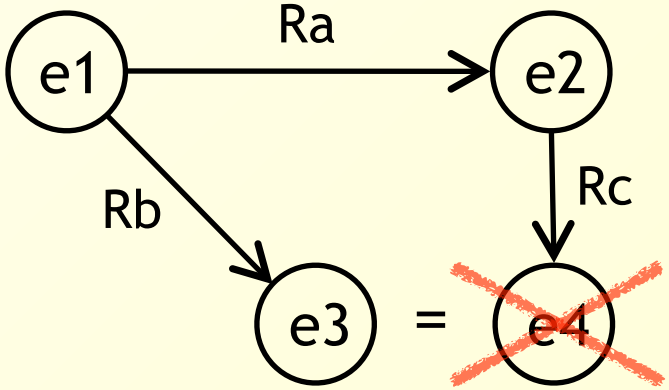
Quelle représentation des connaissances ?

Le Loup, la chèvre et le choux

- **Développer un graphe de recherche en CLIPS :**
  - Les états (situations) sont implémentés par des faits
  - Les transitions (actions) sont implémentés par des règles

Attention aux bouclages (situation déjà générée, voire déjà explorée) :
- *Soit on s'assure de ne pas générer deux faits similaires :*
  - Précondition de la règle de génération (complexifie les règles)
- *Soit on supprime le nouveau fait dès qu'il apparaît dans la base de faits :*
  - Règle de suppression des doublons (valable si pas trop de doublons)





Le Loup, la chèvre et le choux

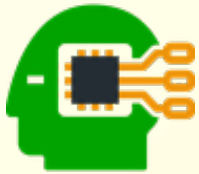
- **Représentation des situations :**
  - Chaque fait représentant une situation doit contenir les emplacements de chacun des protagonistes (le fermier en fait partie).
  - Il n'y a que deux emplacements possibles : GAUCHE et DROITE

```
(deftemplate situation
     (slot loup (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot chevre (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot choux (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot fermier (allowed-values GAUCHE DROITE) (default GAUCHE)))

(deffacts faits-initiaux
     (situation))
```

Les slots contiennent les valeurs par défaut

```
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
For a total of 2 facts.
CLIPS>
```

Le Loup, la chèvre et le choux



fermier1.clp

- **Représentation des actions :**
  - Le fermier ne peut transborder le loup que s'ils sont du même côté :

```
(defrule transborder-loup-GD
     (situation (loup GAUCHE) (chevre ?x) (choux ?y) (fermier GAUCHE))
     =>
     (assert (situation (loup DROITE) (chevre ?x) (choux ?y) (fermier DROITE))))

(defrule transborder-loup-DG
     (situation (loup DROITE) (chevre ?x) (choux ?y) (fermier DROITE))
     =>
     (assert (situation (loup GAUCHE) (chevre ?x) (choux ?y) (fermier GAUCHE))))
```

  - Le fermier doit pouvoir revenir seul sur l'autre rive :

```
(defrule fermier-seul-GD
     (situation (loup ?x) (chevre ?y) (choux ?z) (fermier GAUCHE))
     =>
     (assert (situation (loup ?x) (chevre ?y) (choux ?z) (fermier DROITE))))
```

Combien de règles doit-on écrire avec cette méthode ?

Le Loup, la chèvre et le choux

fermier2.clp

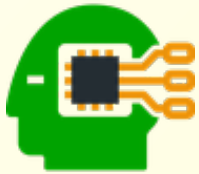
- **Une seule règle de transition par protagoniste :**
  - Méthode pour changer de rive à chaque transition :

```
(deftemplate situation
     (slot loup (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot chevre (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot choux (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot fermier (allowed-values GAUCHE DROITE) (default GAUCHE)))

(deffacts faits-initiaux
     (suivant GAUCHE DROITE)
     (suivant DROITE GAUCHE)
     (situation))

(defrule transborder-loup
     (situation (loup ?rive) (chevre ?x) (choux ?y) (fermier ?rive))
     (suivant ?rive ?new)
     =>
     (assert (situation (loup ?new) (chevre ?x) (choux ?y) (fermier ?new))))

(defrule transborder-chevre
     (situation (loup ?x) (chevre ?rive) (choux ?y) (fermier ?rive))
     (suivant ?rive ?new)
     =>
     (assert (situation (loup ?x) (chevre ?new) (choux ?y) (fermier ?new))))
```

Pour passer de GAUCHE à DROITE et inversement

Combien de règles doit-on écrire avec cette méthode ?

Le Loup, la chèvre et le choux



- **Simplifications des règles de transition :**
  - Possibilité d'effectuer des copies différentielles des faits en ne précisant que les slots modifiés (les autres sont recopiés) :

```
(defrule transborder-loup
     (situation (loup ?rive) (chevre ?x) (choux ?y) (fermier ?rive))
     (suivant ?rive ?new)
     =>
     (assert (situation (loup ?new) (chevre ?x) (choux ?y) (fermier ?new))))
```

```
(defrule transborder-loup
     ?node <- (situation (loup ?rive) (fermier ?rive))
     (suivant ?rive ?new)
     =>
     (duplicate ?node (loup ?new) (fermier ?new)))
```

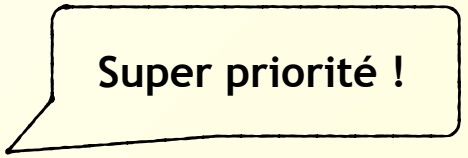
Plus besoin des variables ?x et ?y

Combien de règles doit-on écrire avec cette méthode ?

Le Loup, la chèvre et le choux

fermier3.clp

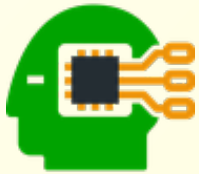
- **Identification de la situation finale :**
  - Tous les protagonistes sont sur la rive droite :




```
(defrule eureka
     (declare (salience 100))
     (situation (loup DROITE) (chevre DROITE) (choux DROITE) (fermier DROITE))
     =>
     (printout t "Eureka" crlf)
     (halt))
```

- **Testons notre programme…**
  - Rien ne va plus !
  - *Il faut passer à une phase de débugage…*
    - (watch facts) permet de visualiser l'évolution de la base de faits
    - (watch rules) permet de visualiser le déclenchement des règles
    - (run 10) demande au moteur d'effectuer 10 cycles

Le Loup, la chèvre et le choux



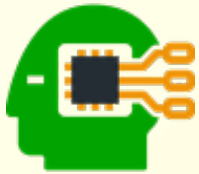
- **Affichage de la trace de l'exécution du programme :**

```
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (reset)
CLIPS> (facts)
f-1     (suivant GAUCHE DROITE)
f-2     (suivant DROITE GAUCHE)
f-3     (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
For a total of 3 facts.
CLIPS> (run 5)
FIRE    1 transborder-loup: f-3,f-1
==> f-4     (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier DROITE))
FIRE    2 transborder-loup: f-4,f-2
==> f-5     (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
FIRE    3 transborder-loup: f-5,f-1
==> f-6     (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier DROITE))
FIRE    4 transborder-loup: f-6,f-2
==> f-7     (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
FIRE    5 transborder-loup: f-7,f-1
==> f-8     (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier DROITE))
CLIPS>
```

Que constatez-vous ?

Quelle solution proposez-vous ?

Le Loup, la chèvre et le choux



fermier4.clp

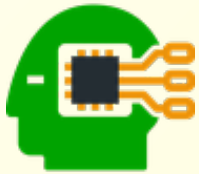
- **Une règle qui supprime les doublons :**
  - Deux faits différents mais qui contiennent les mêmes données
  - On supprime le fait le plus récent (utilisation de fact-index) :

```
(defrule deja-vu
     (declare (salience 100))
     ?f1 <- (situation (loup ?x) (chevre ?y) (choux ?z) (fermier ?w))
     ?f2 <- (situation (loup ?x) (chevre ?y) (choux ?z) (fermier ?w))
     (test (> (fact-index ?f2) (fact-index ?f1)))
     =>
     (retract ?f2))
```

- **Testons notre programme…**

```
FIRE    1 transborder-loup: f-3,f-1
==> f-4     (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier DROITE))
FIRE    2 transborder-loup: f-4,f-2
==> f-5     (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
FIRE    3 deja-vu: f-3,f-5
<== f-5     (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
FIRE    4 fermier-seul: f-4,f-2
==> f-6     (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
...
FIRE   18 eureka: f-16
Eureka
```

Le Loup, la chèvre et le choux



- **Autre méthode pour éviter les doublons :**
  - En configurant le moteur d'inférences pour interdire les doublons :-)

```
CLIPS> (set-fact-duplication FALSE)
```

- **Base de faits en fin d'exécution :**

```
f-3     (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
f-4     (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier DROITE))
f-5     (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
f-6     (situation (loup DROITE) (chevre DROITE) (choux GAUCHE) (fermier DROITE))
f-7     (situation (loup GAUCHE) (chevre DROITE) (choux GAUCHE) (fermier GAUCHE))
f-8     (situation (loup GAUCHE) (chevre DROITE) (choux DROITE) (fermier DROITE))
f-9     (situation (loup GAUCHE) (chevre GAUCHE) (choux DROITE) (fermier GAUCHE))
f-10    (situation (loup DROITE) (chevre GAUCHE) (choux DROITE) (fermier DROITE))
f-11    (situation (loup DROITE) (chevre GAUCHE) (choux DROITE) (fermier GAUCHE))
f-12    (situation (loup DROITE) (chevre DROITE) (choux DROITE) (fermier DROITE))
```

Que constatez-vous ?

Quelle solution proposez-vous ?

Le Loup, la chèvre et le choux

fermier5.clp

- **Identification et suppression des situations "sans avenir" :**

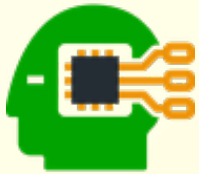
```
(defrule loup-mange-chevre
     (declare (salience 100))
     ?fact <- (situation (loup ?rive) (chevre ?rive) (fermier ~?rive))
     =>
     (retract ?fact))

(defrule chevre-mange-choux
     (declare (salience 100))
     ?fact <- (situation (chevre ?rive) (choux ?rive) (fermier ~?rive))
     =>
     (retract ?fact))
```

- **Base de faits en fin d'exécution :**

```
f-3     (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
f-6     (situation (loup GAUCHE) (chevre DROITE) (choux GAUCHE) (fermier DROITE))
f-7     (situation (loup GAUCHE) (chevre DROITE) (choux GAUCHE) (fermier GAUCHE))
f-9     (situation (loup DROITE) (chevre DROITE) (choux GAUCHE) (fermier DROITE))
f-12    (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE))
f-15    (situation (loup DROITE) (chevre GAUCHE) (choux DROITE) (fermier DROITE))
f-16    (situation (loup DROITE) (chevre GAUCHE) (choux DROITE) (fermier GAUCHE))
f-18    (situation (loup DROITE) (chevre DROITE) (choux DROITE) (fermier DROITE))
```

Notre programme est-il terminé ?

Le Loup, la chèvre et le choux



- **Pour retrouver la suite d'actions il faut construire un arbre :**
  - On va ajouter pour chaque nœud une référence vers son père :

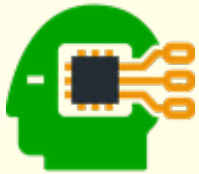
```
(deftemplate situation
     (slot loup (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot chevre (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot choux (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot fermier (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot pere (type FACT-ADDRESS SYMBOL) (default nil))
```

  - Les règles de transition doivent remplir ce nouveau slot :

```
(defrule transborder-loup
     ?fact <- (situation (loup ?rive) (fermier ?rive))
     (suivant ?rive ?new)
     =>
     (duplicate ?fact (loup ?new) (fermier ?new) (pere ?fact)))

(defrule fermier-seul
     ?fact <- (situation (fermier ?rive))
     (suivant ?rive ?new)
     =>
     (duplicate ?fact (fermier ?new) (pere ?fact)))
```

Notre programme est-il terminé ?

Le Loup, la chèvre et le choux



fermier6.clp

- **Il faut pouvoir afficher en clair la suite d'actions :**
  - Chaque fait va contenir la description de l'action qui l'a créé :

```
(deftemplate situation
     (slot loup (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot chevre (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot choux (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot fermier (allowed-values GAUCHE DROITE) (default GAUCHE))
     (slot pere (type FACT-ADDRESS SYMBOL) (default nil))
     (slot action (type STRING) (default "Situation initiale")))

(defrule transborder-loup
     ?fact <- (situation (loup ?rive) (fermier ?rive))
     (suivant ?rive ?new)
     =>
     (duplicate ?fact (loup ?new) (fermier ?new) (pere ?fact)
                      (action "Le fermier passe avec le loup")))

(defrule fermier-seul
     ?fact <- (situation (fermier ?rive))
     (suivant ?rive ?new)
     =>
     (duplicate ?fact (fermier ?new) (pere ?fact)
                      (action "Le fermier revient tout seul")))
```

Le Loup, la chèvre et le choux

- **Testons notre programme…**

```
CLIPS> (reset)
CLIPS> (run)
Eureka
CLIPS> (facts)
f-1  (suivant GAUCHE DROITE)
f-2  (suivant DROITE GAUCHE)
f-3  (situation (loup GAUCHE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE) (action "Situation initiale") (pere nil))
f-5  (situation (loup GAUCHE) (chevre DROITE) (choux GAUCHE) (fermier DROITE) (action "avec la chèvre") (pere <Fact-3>))
f-7  (situation (loup GAUCHE) (chevre DROITE) (choux GAUCHE) (fermier GAUCHE) (action "tout seul") (pere <Fact-5>))
f-8  (situation (loup DROITE) (chevre DROITE) (choux GAUCHE) (fermier DROITE) (action "avec le loup") (pere <Fact-7>))
f-10 (situation (loup DROITE) (chevre GAUCHE) (choux GAUCHE) (fermier GAUCHE) (action "avec la chèvre") (pere <Fact-8>))
f-12 (situation (loup DROITE) (chevre GAUCHE) (choux DROITE) (fermier DROITE) (action "avec le choux") (pere <Fact-10>))
f-13 (situation (loup GAUCHE) (chevre GAUCHE) (choux DROITE) (fermier GAUCHE) (action "avec le loup") (pere <Fact-12>))
f-15 (situation (loup GAUCHE) (chevre DROITE) (choux DROITE) (fermier DROITE) (action "avec la chèvre") (pere <Fact-13>))
f-21 (situation (loup DROITE) (chevre GAUCHE) (choux DROITE) (fermier GAUCHE) (action "tout seul") (pere <Fact-12>))
f-22 (situation (loup DROITE) (chevre DROITE) (choux DROITE) (fermier DROITE) (action "avec la chèvre") (pere <Fact-21>))
For a total of 12 facts.
CLIPS>
```

Comment réaliser l'affichage des actions ?

```
Situation initiale
Le fermier passe avec la chèvre
Le fermier revient tout seul
Le fermier passe avec le loup
Le fermier passe avec la chèvre
Le fermier passe avec le choux
Le fermier revient tout seul
Le fermier passe avec la chèvre
```

Le Loup, la chèvre et le choux

fermier7.clp

- **Fonction récursive d'affichage de la solution :**
  - Cette fonction doit être appelée par la règle "eureka"
  - L'appel récursif doit être placé avant le printout de façon à afficher les différentes actions à partir de la situation initiale :

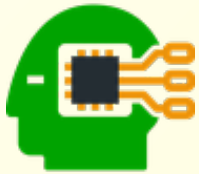
```
(deffunction afficher-solution (?noeud)
     (if (neq ?noeud nil) then
          (bind ?pere (fact-slot-value ?noeud pere))
          (bind ?action (fact-slot-value ?noeud action))
          (afficher-solution ?pere)
          (printout t ?action crlf)))

(defrule eureka
     (declare (salience 100))
     ?fact <- (situation (loup DROITE) (chevre DROITE) (choux DROITE) (fermier DROITE))
     =>
     (afficher-solution ?fact)
     (halt))
```

Définition d'une nouvelle fonction

exécution

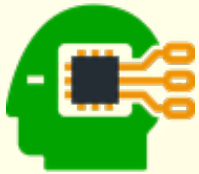
```
CLIPS> (run)
Situation initiale
Le fermier passe avec la chèvre
Le fermier revient tout seul
Le fermier passe avec le loup
Le fermier passe avec la chèvre
Le fermier passe avec le choux
Le fermier revient tout seul
Le fermier passe avec la chèvre
CLIPS>
```

Le Loup, la chèvre et le choux



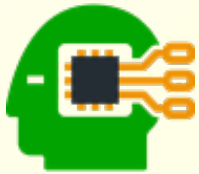
- **Quel bilan peut-on tirer de cet exemple ?**
  - La méthode de résolution (construction d'un graphe de recherche) n'est pas imposée par le moteur d'inférence, elle a été implémentée.
  - Pourquoi Prolog est *a priori* plus adapté pour résoudre ce problème ?
    - La méthode de recherche est déjà implémentée dans le moteur d'inférence
    - Jetons quand-même un petit coup d'oeil sur le code Prolog…
  - Pourquoi je préfère quand-même CLIPS ? (avis personnel)
    - Fonctionnement plus proche de la cognition humaine (méthode déductive).
    - Le code Prolog pour ce problème n'est pas forcément + simple à concevoir…
    - Polyvalence : on peut modifier / adapter cette méthode, développer des approches hybrides (méthodes hypothético-déductives, stratégies heuristiques, etc.).
  - Au final, sur ce problème, les deux approches (chaînage avant vs. chaînage arrière) sont sensiblement équivalentes mais :
    - Une résolution CLIPS est une recherche dans un espace de configurations
    - Une résolution Prolog est une démonstration logique (preuve)

Aspects fonctionnels et impératifs



- **Fonctions sur les multivalués (listes) :**
  - Soit la variable multivaluée `?liste` liée à la valeur (a b c d) :

```
(length$ ?liste)                          → 4
(first$ ?liste)                           → (a)
(rest$ ?liste)                            → (b c d)
(nth$ 2 ?liste)                           → b
(member$ b ?liste)                        → 2
(insert$ ?liste 2 #)                      → (a # b c d)
(insert$ ?liste 2 ?liste)                 → (a a b c d b c d)
(delete$ ?liste 2 3)                      → (a d)
(subseq$ ?liste 2 3)                      → (b c)
(replace$ ?liste 2 3 #)                   → (a # d)
(implode$ ?liste)                         → "a b c d"
(explode$ "a b c d")                      → (a b c d)
(subsetp (create$ b c) ?liste)            → TRUE
```



Aspects fonctionnels et impératifs

- **Fonctions sur les chaînes :**

  `(str-cat <expr>*), (sub-string <int> <int> <expr>),`
  `(str-index <expr> <expr>), (str-compare <expr> <expr>),`
  `(str-length <expr>), (upcase <expr>), (lowcase <expr>)...`

- **Fonctions arithmétiques :**

  `(+ <expr> <expr>+), (- <expr> <expr>+), (* <expr> <expr>+),`
  `(/ <expr> <expr>+), (div <expr> <expr>), (mod <expr> <expr>),`
  `(** <expr> <expr>), (exp <expr>), (log <expr>),`
  `(max <expr>+), (min <expr>+), (abs <expr>), (sqrt <expr>)...`

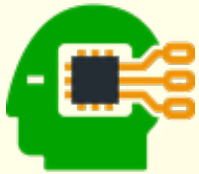
- **Fonctions trigonométriques (25) :**

  `(sin <expr>), (cos <expr>), (tan <expr>), (sinh <expr>)...`

- **Transtypage et conversion :**

  `(float <expr>), (integer <expr>),`
  `(deg-rad <expr>), (rad-deg <expr>)...`

Aspects fonctionnels et impératifs

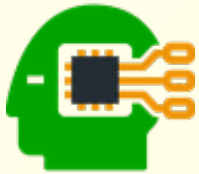


- **Définir de nouvelles fonctions :**

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

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

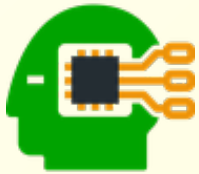
  - Exemple de fonction utilisateur :

```
(deffunction afficher-args (?a ?b $?c)
   (printout t ?a " " ?b " et " (length$ ?c) " extras: " ?c crlf)

   CLIPS> (afficher-args 1 2)
   1 2 et 0 extras: ()
   CLIPS> (afficher-args a b c d)
   a b et 2 extras: (c d)
```



Aspects fonctionnels et impératifs

- **Opérations sur les fichiers :**
  - `(open <file-name> <logical-name> [<mode>])`
    `"r" (lecture seule) "w" (écriture seule)`
    `"r+" (lecture et écriture) "a" (ajout)`
  - `(close [<logical-name>])`
  - `(rename <old-file-name> <new-file-name>)`
  - `(remove <file-name>)`
  - `(dribble-on <file-name>) (dribble-off)`
- **Lecture et écriture dans un flux d'entrée/sortie :**
  - `(read [<logical-name>])`
  - `(readline [<logical-name>])`
  - `(printout <logical-name> <expression>*)`
  - `(format <logical-name> <string> <expression>*)`
    - cf. printf du langage C



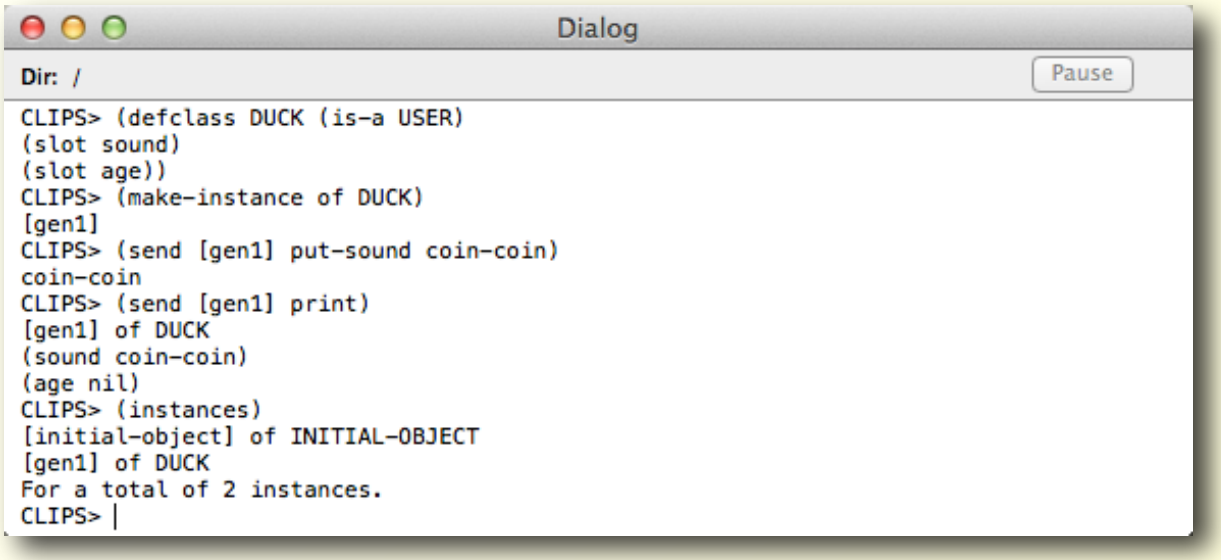
Aspects fonctionnels et impératifs

- Liaison variable-expression :
  - `(bind <variable> <expression>)`
- Si ... alors ... sinon :
  - `(if <expression> then <action>* [else <action>*])`
- Boucle tant que :
  - `(while <expression> [do] <action>*)`
- Boucle pour :
  - `(loop-for-count <range> [do] <action>*)`
  - `<range> ::= <end-index> | (<variable> [<start> <end>])`
- Boucle pour chaque :
  - `(progn$ (<variable> <expression>) <expression>*)`
- Sélectionner selon :
  - `(switch <test> (case <expression> then <action>*)+)`

La programmation orientée objet

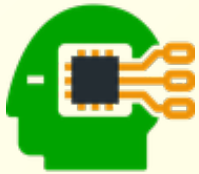
- **Les objets de la couche COOL :**
  - Abstraction + encapsulation + polymorphisme + héritage multiple
  - Une classe peut être abstraite ou concrète, réactive ou non
  - Les attributs sont monovalués ou multivalués

```
(defclass <classe> (is-a <super-classe>+)

     (role abstract|concrete)
     [(pattern-match reactive)]
     (slot|multislot <attribut> ... )*)
```




```
CLIPS> (browse-classes)
OBJECT
  PRIMITIVE
    NUMBER
      INTEGER
      FLOAT
    LEXEME
      SYMBOL
      STRING
    MULTIFIELD
    ADDRESS
      EXTERNAL-ADDRESS
      FACT-ADDRESS
      INSTANCE-ADDRESS *
    INSTANCE
      INSTANCE-ADDRESS *
      INSTANCE-NAME
  USER
    INITIAL-OBJECT
    DUCK
CLIPS>
```

The Java Expert System Shell



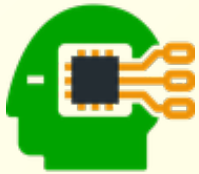
- **Jess est une implémentation de CLIPS en Java :**

  - **On peut accéder à Jess depuis les méthodes Java :**
    - On peut créer et manipuler des faits en Java
    - On peut contrôler le moteur d'inférences en Java

  - **On peut accéder à Java depuis les règles Jess :**
    - On peut instancier des objets Java dans une règle
    - On peut réaliser des envois de messages dans une règle

  - **On peut associer des faits Jess aux objets Java :**
    - On peut donc effectuer des inférences sur des objets Java
    - On peut "mettre de l'intelligence" dans un programme Java

Jess : *executeCommand*



Test_1.java

- **Instancier Jess et exécuter une commande :**

```
import jess.*;

public class Test {

     public Test() {
          try {
               Rete moteur = new Rete();
               moteur.executeCommand("(batch test.clp)");
               moteur.executeCommand("(une_commande)");
          }
          catch (JessException e) {
               e.printStackTrace();
          }
     }

     public static void main(String[] argv) {
          new Test();
     }
}
```

```
(deffunction une_commande ()
     (printout t "Hello World !" crlf))
```

- javac -classpath ".:../jess.jar" Test.java
- java -classpath ".:../jess.jar" Test

Jess : *executeCommand*

Test_2.java

- **Récupérer le résultat d'une commande Jess :**

```
public Test() {
     try {
          Rete moteur = new Rete();
          moteur.executeCommand("(deffunction square (?n) (return (* ?n ?n)))");
          Value value = moteur.executeCommand("(square 3)");
          System.out.println(value.intValue(moteur.getGlobalContext()));
     }
     catch (JessException e) { e.printStackTrace(); }
}
```

  - Encapsulation des données :
    - Toute donnée de Jess est stockée dans un objet de type **`jess.Value`** :

value.type()	Type Jess	Type Java	Récupération de la valeur dans Java
2	jess.RU.STRING	String	value.stringValue(moteur.getGlobalContext)
4	jess.RU.INTEGER	int	value.intValue(moteur.getGlobalContext)
32	jess.RU.FLOAT	double	value.floatValue(moteur.getGlobalContext)
...	...	...	...

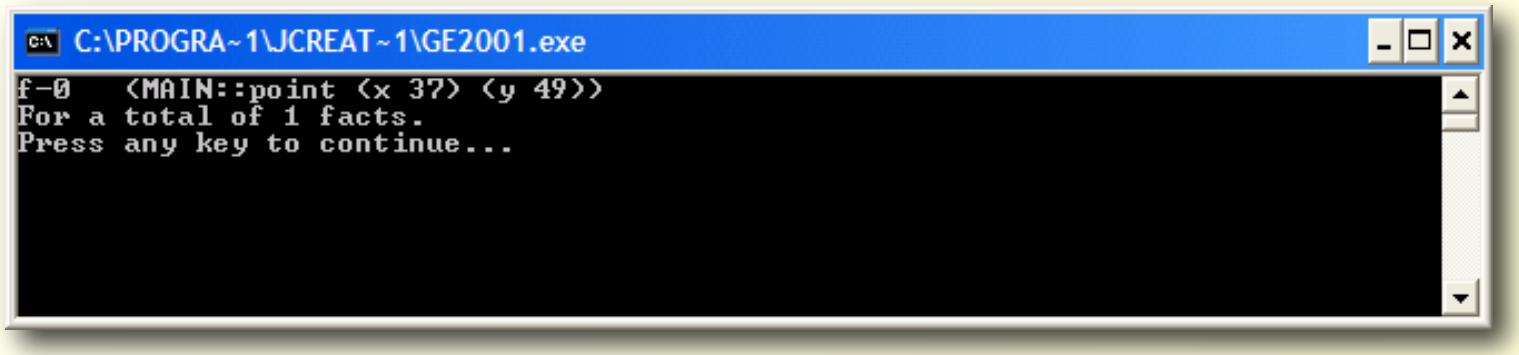
    - La classe **`jess.Context`** représente un contexte d'évaluation pour la résolution des variables et les appels de fonction

Jess : *assertFact*

Test_3.java

- Créer un fait structuré depuis Java :

```
public Test() {
     try {
          Rete moteur = new Rete();
          moteur.executeCommand("(deftemplate point (slot x) (slot y))");

          Fact fact = new Fact("point", moteur);
          fact.setSlotValue("x", new Value(37, RU.INTEGER));
          fact.setSlotValue("y", new Value(49, RU.INTEGER));
          moteur.assertFact(fact);

          moteur.executeCommand("(facts)");
     }
     catch (JessException e) { e.printStackTrace(); }
}
```

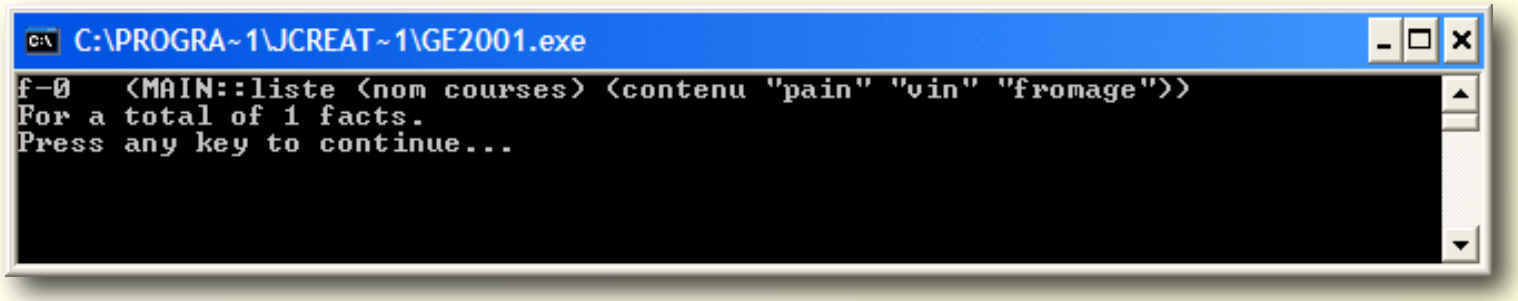


Jess : *assertFact*

Test_4.java

- **Créer un fait structuré depuis Java (suite) :**

```
Rete moteur = new Rete();
moteur.executeCommand("(deftemplate liste (slot nom) (multislot contenu))");

ValueVector vector = new ValueVector();
vector.add(new Value("pain", RU.STRING));
vector.add(new Value("vin", RU.STRING));
vector.add(new Value("fromage", RU.STRING));

Fact fact = new Fact("liste", moteur);
fact.setSlotValue("nom", new Value("courses", RU.ATOM));
fact.setSlotValue("contenu", new Value(vector, RU.LIST));
moteur.assertFact(fact);

moteur.executeCommand("(facts)");
```

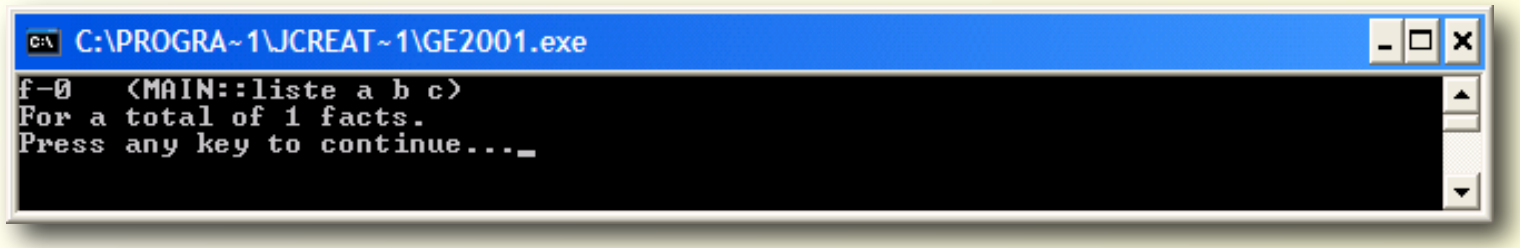


Jess : *assertFact*

Test_5.java

- **Créer un fait ordonné depuis Java :**

```
Rete moteur = new Rete();

ValueVector vector = new ValueVector();
vector.add(new Value("a", RU.ATOM));
vector.add(new Value("b", RU.ATOM));
vector.add(new Value("c", RU.ATOM));

Fact fact = new Fact("liste", moteur);
fact.setSlotValue("__data", new Value(vector, RU.LIST));
moteur.assertFact(fact);

moteur.executeCommand("(facts)");
```



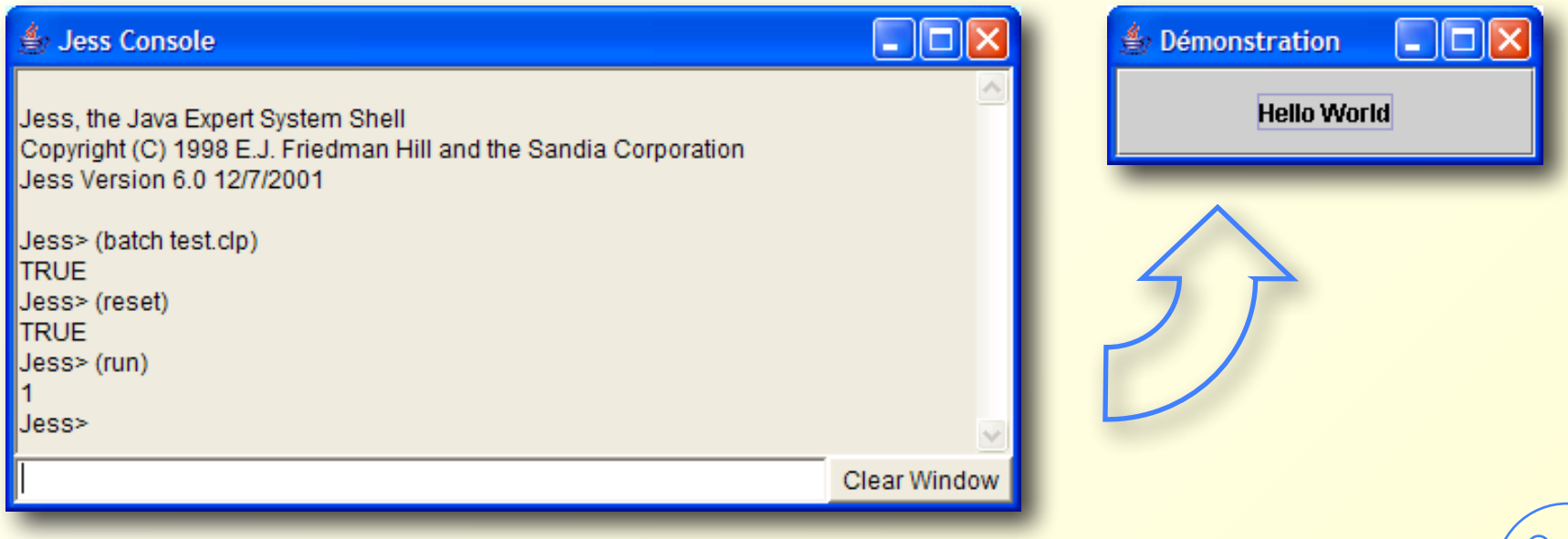
- Dans Jess, les faits ordonnés sont implémentés à l'aide de faits structurés ayant un unique slot multivalué : **__data**
- Il n'y a pas besoin de créer de template

Jess : *java reflection*

Hello.clp

- **Instancier des objets Java depuis Jess :**

```
(defrule test "crée et ouvre une fenêtre Swing"

     =>
     (bind ?frame (new javax.swing.JFrame "Démonstration"))
     (bind ?button (new javax.swing.JButton "Hello World"))
     (call (get ?frame "contentPane") add ?button)
     (call ?frame pack)
     (set ?frame visible TRUE))
```



Jess : *java reflection*

Hello.clp

- **Ajout de deux listeners :**

```
(deffunction say-hello (?evt)
    (printout t "Hello, World!" crlf))

(deffunction frame-handler (?evt)
    (if (= (?evt getID) (get-member ?evt WINDOW_CLOSING)) then
        (call (get ?evt source) dispose)
        (exit)))

(defrule test "crée et ouvre une fenêtre Swing"

    =>
    (bind ?frame (new javax.swing.JFrame "Démonstration"))
    (bind ?button (new javax.swing.JButton "Hello World"))
    (call (get ?frame "contentPane") add ?button)

    (call ?frame addWindowListener (new jess.awt.WindowListener frame-handler (engine)))
    (call ?button addActionListener (new jess.awt.ActionListener say-hello (engine)))

    (call ?frame pack)
    (set ?frame visible TRUE))
```

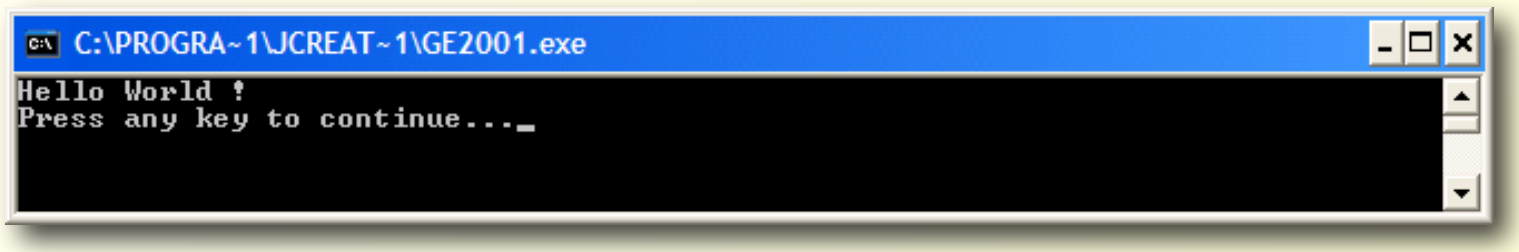
Jess : *store-fetch*

- Transférer une donnée de Jess vers Java :

test.clp

```
(defrule test
     =>
     (store CHAINE "Hello World !")
)
```

```
     try
     {
          moteur = new Rete();
          moteur.executeCommand("(batch test.clp)");
          moteur.executeCommand("(reset)");
          moteur.executeCommand("(run)");

          String chaine = moteur.fetch("CHAINE").stringValue(moteur.getGlobalContext());
          System.out.println(chaine);
     }
     catch (JessException e)
     {
          e.printStackTrace();
     }
```



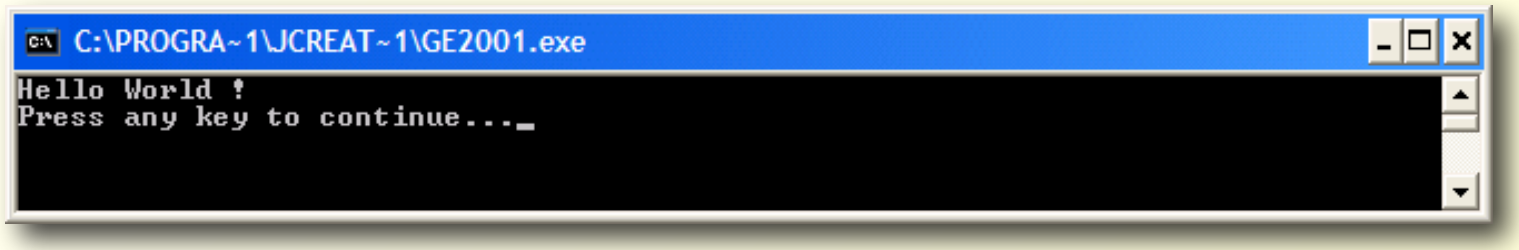
Jess : *store-fetch*

- Transférer une donnée de Java vers Jess :

test.clp

```
(defrule test
     =>
     (printout t (call (fetch CHAINE) toString) crlf)
)
```

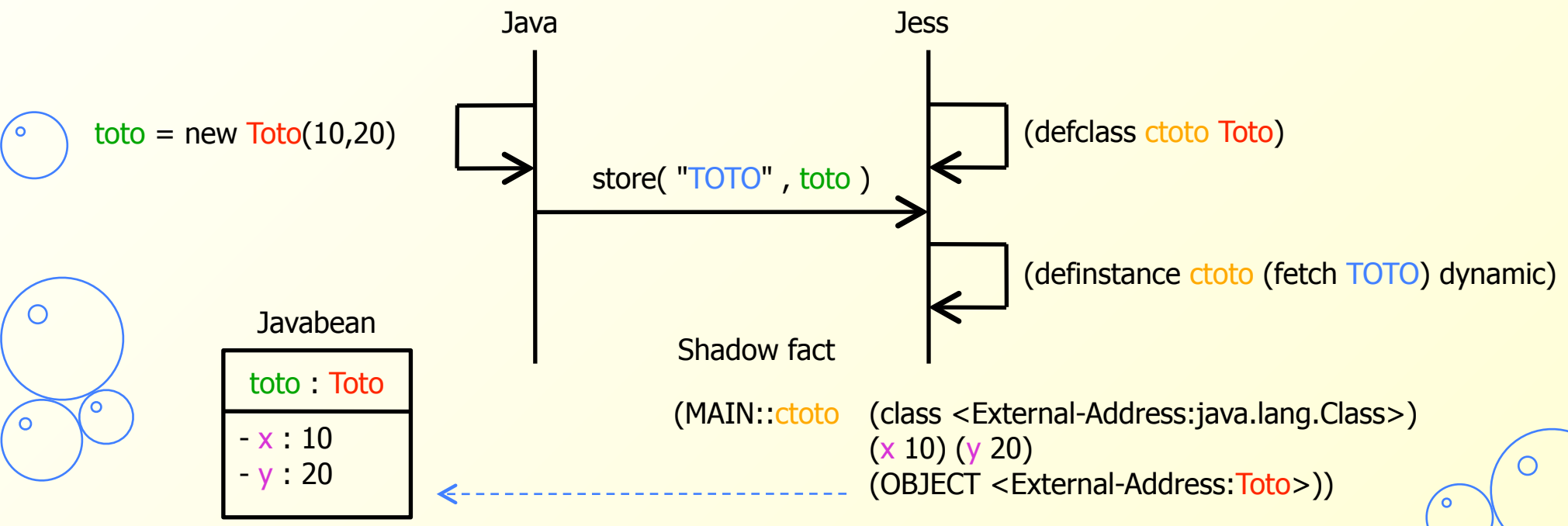
```
     try
     {
          moteur = new Rete();
          moteur.store("CHAINE", "Hello World !");
          moteur.executeCommand("(batch test.clp)");
          moteur.executeCommand("(reset)");
          moteur.executeCommand("(run)");
     }
     catch (JessException e)
     {
          e.printStackTrace();
     }
```



Jess : *shadow facts*

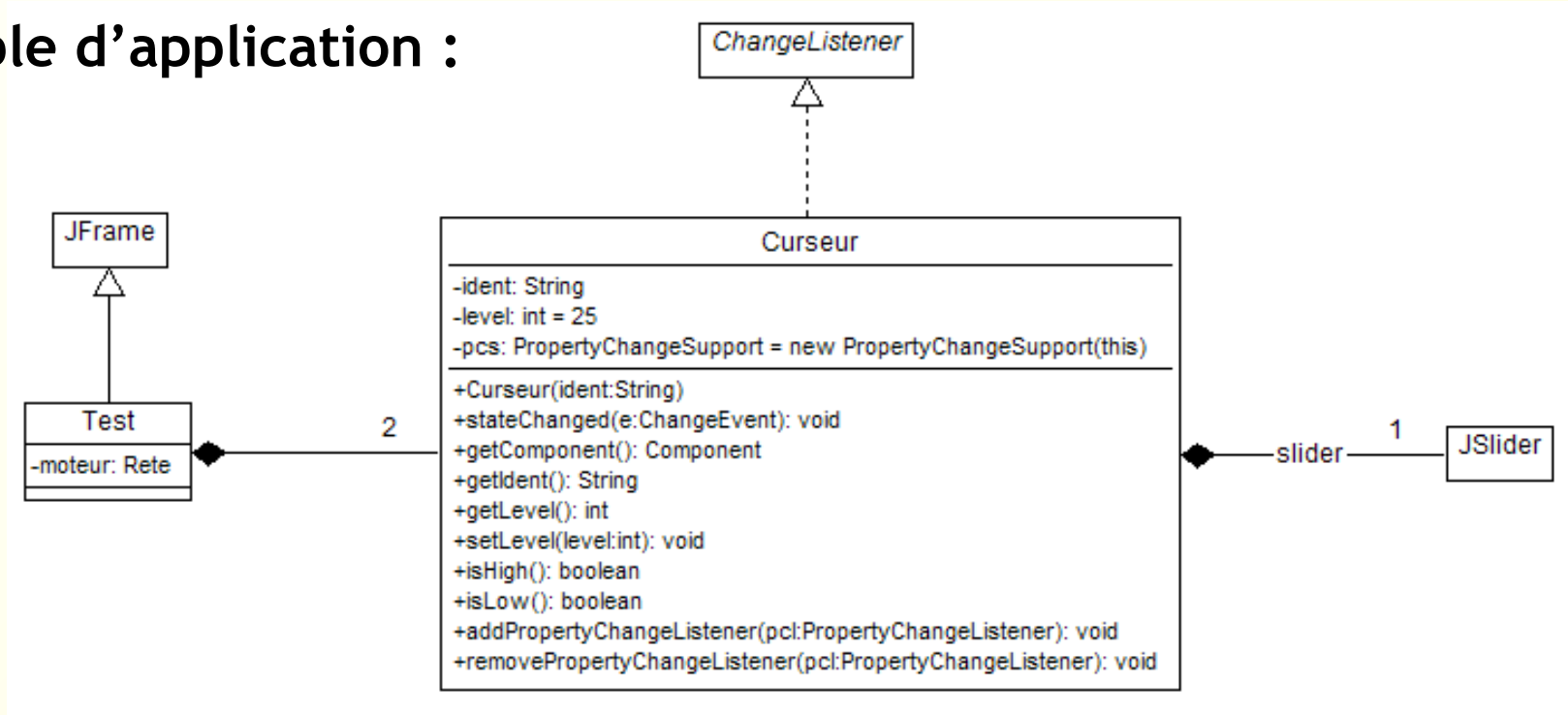
- **Créer un « shadow fact » à partir d'un objet Java :**

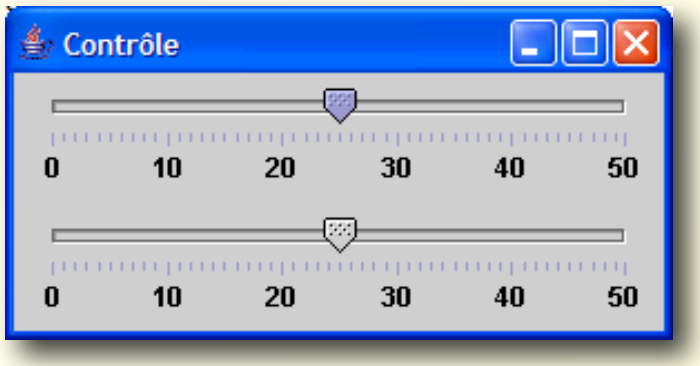
  - Le principe repose à la fois sur l'utilisation des *Javabeans* et du *PropertyChangeSupport* qui permet de générer un évènement lorsque la valeur d'un attributs est modifié
  - L'objet doit être préalablement enregistré auprès du moteur :



Jess : *shadow facts*

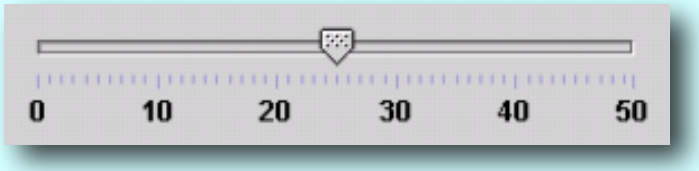
- **Exemple d'application :**





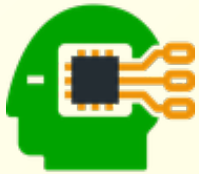
Jess : *shadow facts*

- Codage du Javabean Curseur (partie graphique) :




```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Curseur implements ChangeListener
{
     private String ident;
     private JSlider slider;
     private int level = 25;

     public Curseur(String ident)
     {
          this.ident = ident;
          slider = new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
          slider.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
          slider.setMajorTickSpacing(10);
          slider.setMinorTickSpacing(1);
          slider.setPaintTicks(true);
          slider.setPaintLabels(true);
          slider.addChangeListener(this);
     }

     ...
```

Jess : *shadow facts*



- **Suite du codage du Javabean Curseur :**

```
public Component getComponent() { return slider; }
public String getIdent() { return ident; }
public int getLevel() { return level; }
public void setLevel(int level)
{
     slider.setValue(level);
}
```

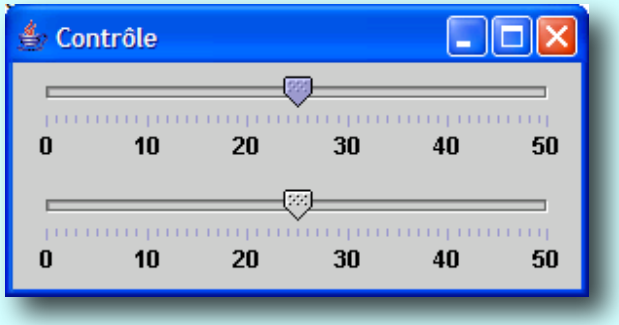
Getters et Setters

```
public void stateChanged(ChangeEvent e)
{
     JSlider source = (JSlider)e.getSource();
     if (!source.getValueIsAdjusting())
     {
          int oldlevel = level;
          level = (int)source.getValue();
          pcs.firePropertyChange("level", new Integer(oldlevel), new Integer(level));
     }
}

private PropertyChangeSupport pcs = new PropertyChangeSupport(this);
public void addPropertyChangeListener(PropertyChangeListener pcl)
{
     pcs.addPropertyChangeListener(pcl);
}
public void removePropertyChangeListener(PropertyChangeListener pcl)
{
     pcs.removePropertyChangeListener(pcl);
}
```

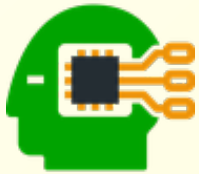
Jess : *shadow facts*

- Codage de la classe Test (partie graphique) :

```
public class Test extends JFrame
{
     Rete moteur;

     public Test()
     {
          super("Contrôle");

          Curseur curseur1 = new Curseur("curseur1");
          Curseur curseur2 = new Curseur("curseur2");

          getContentPane().setLayout(new GridLayout(2, 1));
          getContentPane().add(curseur1.getComponent());
          getContentPane().add(curseur2.getComponent());
          setBounds(100, 100, 300, 150);
          setVisible(true);

          addWindowListener(new WindowAdapter()
          {
               public void windowClosing(WindowEvent e)
               {
                    System.exit(0);
               }
          });

          ...
     }
}
```



Jess : *shadow facts*



- **Suite du codage du constructeur Test :**

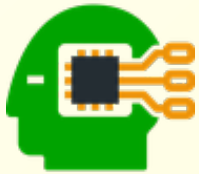
```
try
{
     moteur = new Rete();
     moteur.store("CUR1", curseur1);
     moteur.store("CUR2", curseur2);
     moteur.executeCommand("(batch curseurs.clp)");
     moteur.executeCommand("(reset)");
     moteur.executeCommand("(run)");
}
catch (JessException e)
{
     e.printStackTrace();
}
```

- Autre solution pour enregistrer les deux curseurs :

```
private Funcall f;

f = new Funcall("definstance", moteur);
f.add(new Value("curseur", RU.ATOM));
f.add(new Value(curseur1));
f.execute(moteur.getGlobalContext());

f = new Funcall("definstance", moteur);
f.add(new Value("curseur", RU.ATOM));
f.add(new Value(curseur2));
f.execute(moteur.getGlobalContext());
```

Revient au même mais :
- sans utiliser "store-fetch"
- sans faire de "definstance" côté Jess

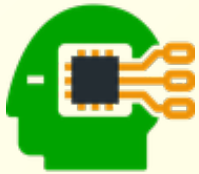
Jess : *shadow facts*



- Base de connaissances Jess :

curseurs.clp

```
(defclass curseur Curseur)
(definstance curseur (fetch CUR1) dynamic)
(definstance curseur (fetch CUR2) dynamic)


(deffacts idle-fact (idle 0))


(defrule affichage "Pour afficher la valeur d'un curseur après modification"

     (declare (salience 10))
     (curseur (ident ?ident) (level ?level))
     =>
     (printout t "Valeur du " ?ident " = " ?level crlf))


(defrule reaction "Pour ajuster l'autre curseur en fonction du curseur modifié"

     (curseur (ident ?ident) (level ?level))
     (curseur (ident ~?ident) (OBJECT ?curseur))
     =>
     (call ?curseur setLevel ?level))

(defrule attente "Pour faire tourner le moteur en boucle si rien ne se passe"

     (declare (salience -999))
     ?idle <- (idle ?n)
     =>
     (retract ?idle)
     (call Thread sleep 100)
     (assert (idle (+ ?n 1))))
```



Jess : *user functions*

- **Ajouter des commandes à Jess :**
  - En définissant un package utilisateur…

```
import jess.*;


public class JessUserFunctions implements Userpackage
{
     public void add(Rete moteur)
     {
          moteur.addUserfunction(new Majuscule());
     }


     public class Majuscule implements Userfunction
     {
          public String getName()
          {
               return "majuscule";
          }


          public Value call(ValueVector values, Context context) throws JessException
          {
               return new Value(values.get(1).stringValue(context).toUpperCase(), RU.STRING);
          }
     }
}
```

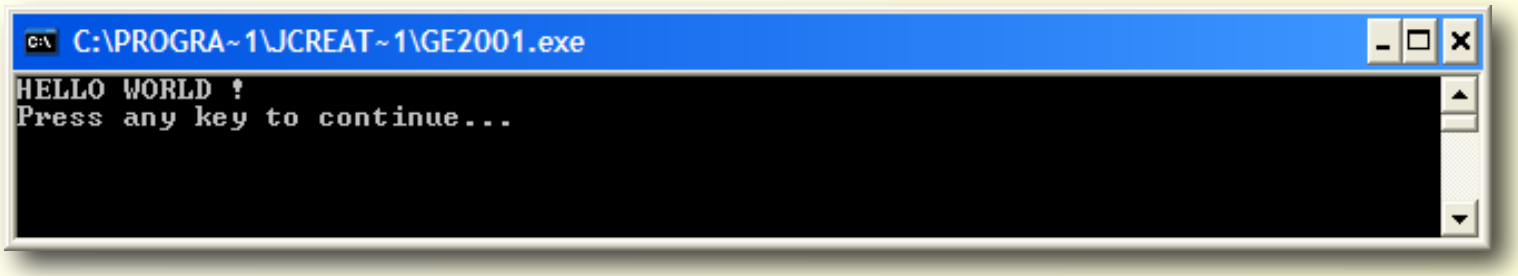
Jess : *user functions*

- Ajouter des commandes à Jess (suite) :

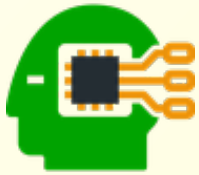
```
public class Test extends Rete
{
     public Test() throws JessException
     {
          addUserpackage(new JessUserFunctions());
          executeCommand("(batch test.clp)");
          executeCommand("(reset)");
          executeCommand("(run)");
     }
}
```

```
(defrule test
     =>
     (printout t (majuscule "hello world !") crlf))
```

test.clp



Le wrapper PyCLIPS



- **PyCLIPS est une interface d'accès à CLIPS en Python 2 :**
  - Le compilateur gcc doit être installé sur la machine hôte :

```
> tar -zxvf pyclips-1.0.7.tgz
> cd pyclips
> sudo python setup.py build
> sudo python setup.py install
```

  - Définitions de nouvelles fonctions CLIPS écrites en Python :

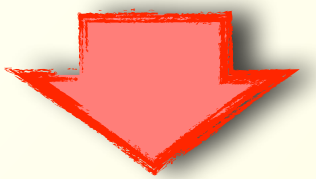
```
import clips

def print_console(texte):
        print texte

clips.RegisterPythonFunction(print_console)
clips.BuildFunction(
        "print_console",
        "?texte",
        "(python-call print_console ?texte)")
```

Le wrapper PyCLIPS

- Chargement et exécution du code CLIPS en Python :

```
import clips
clips.Load('exemple.clp')
clips.Reset()
clips.Assert('(borne 3)')
clips.Run()
clips.PrintFacts()
```

```
(deffacts initial-facts
          (nombre 1))

(defrule addition
          (borne ?max)
          (nombre ?x&:(< ?x ?max))
          =>
          (print_console ?x)
          (assert (nombre (+ ?x 1))))
```



```
> python exemple.py
1
2
f-0     (initial-fact)
f-1     (nombre 1)
f-2     (borne 3)
f-3     (nombre 2)
f-4     (nombre 3)
For a total of 5 facts.
```