Systèmes Intelligents NFP213
Année 2011-2012

Le langage de Systèmes experts JESS

F.-Y. Villemin (f-yv@cnam.fr)

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

http://dept25.cnam.fr/SIC/index.html

Jess

La documentation en ligne se trouve à l'adresse :

```
http://herzberg.ca.sandia.gov/jess/docs/60/
```

ou encore sur

```
http://herzberg.ca.sandia.gov/jess/manual.pdf
```

Le programme est un fichier `jar` exécutable :

Décompressez le fichier, puis lancez le programme avec :

```
java -classpath jess.jar jess.Main
```

ou

```
java -classpath jess.jar jess.Main fichier.clp
```

Classpath pour Windows XP et JessWin :

```
SET PATH=%PATH%;"C:\Program Files\j2sdk1.5.X\bin"
SET CLASSPATH=.;C:\Jess61p4\jess.jar
CD \Jess61p4\JessWin
Java JessWin
```


Les commandes

Toutes les commandes sont des fonctions entre parenthèses

Commandes de base:

`(batch "nom-de-fichier.clp ")` : charge une base

`(exit)` : pour sortir du programme

`(reset)` : pour préparer les déductions (lecture des faits, fixer un fait initial qui permettra de lancer la machine, repérage des règles déclenchables, oubli des faits appris)

`(run)` : pour lancer la déduction.

`(facts)` : pour voir la liste des faits (et leur numéro)

`(watch all)` : pour obtenir un maximum de verbosité lors des déductions (faits déduits, règles applicables règles appliquées ...)

Les versions `(watch rules)` `(watch compilations)` `(watch activations)` et `(watch facts)` sont un peu moins loquaces…

Composants de base

- Atomes : des noms d'identificateurs
- Nombres : syntaxe standard (entiers double ...)
- Chaînes : syntaxe standard
- Commentaires : précédés d'un point virgule, courent jusqu'à la fin de la ligne
- Fonction : une liste, dont le premier élément est le nom de la fonction les arguments suivants en sont les paramètres : `(+ 5 3)`

  Jess évalue et retourne la valeur calculée (ici 8)
- Impression d'un message sur l'écran :

  `(printout t "texte" 42 "texte" crlf)`

  le `t` au départ et le `crlf` à la fin sont nécessaires

Variables

Pour utiliser une variable dans une fonction, il faut utiliser un identificateur précédé d'un point d'interrogation : ?x

Pour donner une valeur à une variable on utilise la fonction bind :

```
Jess>(bind ?x 123)
123
Jess>?x
123
```


Fonctions

Définition de fonctions:

```
(deffunction <function-name> [<doc-comment>]
(<parameter>*)
<expr>*
[<return-specifier>])
```

Exemple :

```
Jess> (deffunction maximum (?a ?b)
  (if (> ?a ?b) then
       (return ?a)
  else
       (return ?b)))
```

On peut ensuite l'appeler partout dans le programme :

```
(maximum 3 5)
```


Faits simples

Les faits sont des listes caractérisées par leur premier élément :

```
(habile singe)
(proche singe sol)
(poids chaise 25)
```

On utilise la fonction assert pour ajouter un ou des faits à la base :

```
(assert (sur singe sol) (proche banane chaise))
```

La fonction retract permet d'enlever un fait de la base :

```
(retract #fact)
```

Retire de la base le fait numéro #fact (on peut connaître le numéro d'un fait que l'on veut retirer de la base par fact-id)

Faits structurés

Pour définir des faits contenant plusieurs champs, il faut définir la structure comme un template :

```
(deftemplate <deftemplate-name> [extends <classname>]
  [<doc-comment>]
  [(slot <slot-name> [(default | default-dynamic
  <value>)][(type <typespec>)]*)
```

Jess recommande de ne pas utiliser la définition de type

Les slots sont les noms des différents champs

Exemple :

```
(deftemplate automobile
   "Une voiture typique."
   (slot constructeur)
   (slot modele)
   (slot annee (type INTEGER))
   (slot couleur (default blanche)))
```

Faits structurés

On peut ajouter des faits structurés à la base avec `assert` :

```
(assert (automobile (constructeur Renault) (modele
 Clio) (annee 2012)))
```

Cette voiture est blanche par défaut

On peut modifier la valeur d'un champ d'un fait structuré existant avec la fonction `modify` :

```
Jess> (assert (automobile (constructeur Renault)
(modele Clio)(annee 2012)))
<Fact-1>
Jess>(modify #fact(modele Megane) (couleur rouge))
```

Possible parce que l'on sait que le fait a le numéro `#fact`

Définir tous les faits initiaux

Il serait laborieux de construire les faits à partir de la fonction `assert`, la fonction `deffacts` permet de définir tous les faits initiaux en une fois :

```
(deffacts mes-faits "La base de faits"
(habile singe)
(sur singe sol)
…
(inerte chaise)
(proximite (un banane) (deux sol) (verite FALSE))
)
```

Ces faits seront introduits dans la base à chaque commande `reset`

Règles

Une règle est définie par la fonction `defrule` :

```
(defrule nom "commentaire"
premisse1
premisse2
…
premisseN
=>
action1
action2
…
actionP
)
```


Règles

La règle se déclenchera si chaque prémisse peut être unifiée avec un fait ou si une condition est vérifiée

Exemple :

```
(defrule r3
(deplacer ?X ?Y ?Z)
(proximite (un ?Z) (deux sol) (verite FALSE))
=>
(assert (sous ?Y ?Z))
```

se déclenchera si X peut déplacer Y vers Z et si Z n'est pas proche du sol

dans ce cas on en déduira que Y est sous Z (fait rajouté à la base)

L'utilisation de variables (commençant par des points d'interrogation) qui permet de relier entre elles les deux prémisses par une condition

Règles

Autre exemple

```
(defrule r4
(explore ?x ?y FALSE)
(test (courant_d_air ?x ?y))
=>
(printout t "Un courant d'air en " ?x " "?y crlf)
(assert (explore ?x ?y TRUE) (vent ?x ?y))
```

Ici la règle se déclenchera si la première prémisse peut s'unifier avec un fait connu, par contre, la deuxième condition est un test, qui permettra d'activer la règle si la fonction `courant_d_air` retourne le booléen `TRUE`

Avant d'être utilisées dans la fonction `test`, les variables `?x` et `?y` doivent avoir été unifiées avec des faits, il est donc important que les deux "prémisses" soient définies dans cet ordre!

Règles

On peut fixer des conditions sur les variables apparaissant en partie gauche d'une règle :

- Un même nom de variable représente le même objet
- La différence `(sous ?Y ?Z&~?X)`: la prémisse sera vérifiée quand `?Z` sera différent de `?X`
- Le résultat (booléen) d'un appel de fonction `(sous ?Y ?Z&: (> ?Z 10))` se déclenchera si `?Z` est supérieur à `10` (à condition que le deuxième argument de sous soit un nombre!)

Exemple regroupant plusieurs possibilités :

```
Jess> (defrule exemple-3
   (not-b-and-c ?n1&~?b ?n2&~c)
   (different ?d1 ?d2&~?d1)
   (same ?s ?s)
   (more-than-one-hundred ?m&:(> ?m 100))
   (red-or-blue red|blue)
   =>
   (printout t "Found what I wanted!" crlf))
```


Règles

On peut utiliser les connecteurs logiques pour combiner les prémisses

```
Jess> (defrule or-exemple-1
(or (a) (b) (c))
=>)
Jess> (defrule or-exemple-2
(and (or (a)
(b))
(c))
=>)
Jess> (defrule exemple-3
(personne ?x)
(not (marie ?x))
=>
(printout t ?x " n'est pas marie!" crlf))
```


Règles

La règle `rule or-exemple-1` a peu d'intérêt (on suppose par défaut que l'on fait la conjonction des prémisses) mais permet des combinaisons quelconques d'opérateurs booléens

Dans la règle `rule exemple-3`, la variable `?x` a besoin d'être d'abord substituée dans `personne` avant de pouvoir être utilisée dans `not`

La fonction `test` permet de prendre des décisions en fonction de la valeur d'une fonction :

```
Jess> (defrule rule-1
(foo ?X)
(test (> ?X 3))
=>)
```

déclenchera la règle si `?X` vérifie `foo` et si `?X` est plus grand que `3`

Utiliser Jess

Pour un problème donné, il vous faudra :

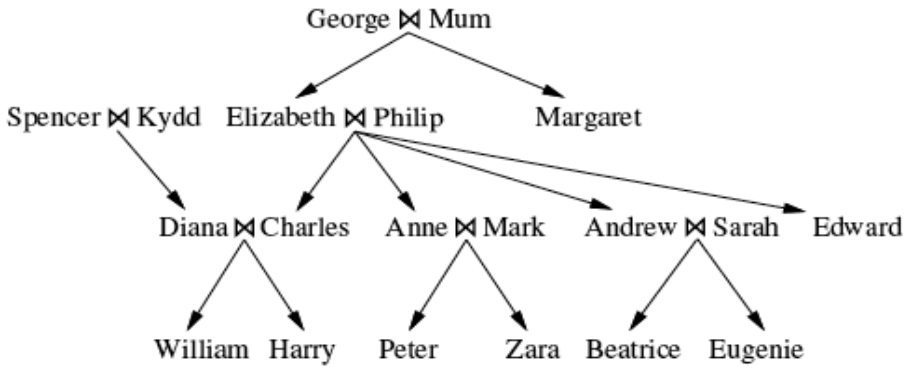
- Déterminer le langage du problème
- Définir les fonctions auxiliaires, les structures éventuelles de faits
- Définir les faits initiaux
- Définir les règles

Ensuite vous pouvez tester votre système dans Jess

- Lancer `Jess`
- Charger votre fichier : `(batch "fichier")`
- Demander qu'il affiche ce qu'il fait : `(watch all)`
- Préparer le système : `(reset)`
- Lancer: `(run)`

Problème

La famille royale britannique :



Problème

Question 1 : Construire le fichier famille.clp, avec les faits de base :

- Etre le père de
- Etre la mère de

Question 2 : Construisez une règle permettant d'obtenir les faits :

- Etre le grand-père de
- Etre la grand-mère de

Question 3 : Construire une règle qui remplace les faits Etre père de et Etre mère de par Etre un parent de

Question 4 : Même question avec :

- Etre oncle ou tante de
- Etre un cousin de
- Etre un ancêtre de
- Appartenir à la famille royale

Question 5 : Construire une règle pour repérer les individus qui n'ont pas de parent dans l'arbre

Chaînage arrière

Jess a un mode "backward-chaining" de Chaînage arrière :

```
(do-backward-chaining factorielle)

(defrule print-factorielle-#
    (factorielle # ?x)
    =>
    (printout t "La factorielle de # est " ?x "."crlf))

(defrule do-factorielle
    (need-factorielle ?x ?r)
    =>
     ;Calcul de la factorielle de ?x dans ?r
    (bind ?r 1)
    (bind ?n ?x)
    (while (> ?n 1)
        (bind ?r (* ?r ?n))
        (bind ?n (- ?n 1)))
    (assert (factorielle ?x ?r)))
```

Défaut

Le raisonnement par défaut peut être réaliser en Jess par le moyen des prédicats "abnomal" de Shoham ("non_voler") :

```
(assert (oiseau Lola)) ; Lola est un oiseau

(defrule défaut-vole ; par défaut, ; par défaut, un oiseau vole
   (oiseau ?x)
   (not (non_voler ?x))
    =>
   (assert (voler ?x)))

(defrule règle-autruche ; une autruche ne vole pas
   (autruche ?x)
    =>
   (assert (non_voler ?x)))

(defrule défaut-vole-retrait ; si un oiseau ne vole pas
   (non_voler ?x)
   ?y <- (voler ?x)
   =>
   (retract ?y)
   (printout t ?y " est retiré" crlf))

(assert (autruche Lola)) ; Lola est une autruche
```


Defquery

Le constructeur Defquery permet d'effectuer des requêtes dans la base de faits :

```
(defquery question
    (declare (variables ?X))
    (foo ?X ?Y))

(deffacts data
    (foo bleu rouge)
    (bar bleu vert)
    (foo bleu rose)
    (foo rouge bleu)
    (foo bleu bleu)
    (foo orange jaune)
    (bar bleu mauve))
```


Defquery

```
(defrule test
   =>
   (bind ?iterator (run-query question bleu))
   (while (?iterator hasNext)
   (bind ?token (call ?iterator next))
   (bind ?fact (call ?token fact 1))
   (bind ?slot (fact-slot-value ?fact __data))
   (bind ?datum (nth$ 2 ?slot))
   (printout t ?fact " " ?slot " " ?datum crlf)))
```

- L'itération n'est pas une fonctionnalité de Jess mais de Java !
- C'est un exemple d'utilisation de la "Java reflection"

Defadvice

Le constructeur Defadvice permet d'exécuter du code avant et/ou après un appel de fonction :

```
Jess> (defadvice before + (bind $?argv (create$ $?
argv 1)))
TRUE
Jess> (+ 2 2)
5
Jess> (defadvice before + (return 1))
TRUE
Jess> (+ 2 2)
1
```