IA01 : Rapport du TP3

Auteurs :
Camille Gerin-Roze
Thomas Perrin

Le 3 Janvier 2016

Table des matières

1 Connaissances nécessaires au système expert **3**

2 Implémentation du système expert **4**
2.1 Choix de la représentation Lisp 4
2.1.1 Base de règles 4
2.1.2 Base de faits 5
2.2 Le moteur d'inférences 5
2.2.1 Validation des prémisses 6
2.2.2 Calcul des nouvelles valeurs 6
2.2.3 Moteur à chaînage avant 6
2.2.4 Test du moteur d'inférences 6

3 Scénarios d'utilisation **7**

Introduction

Chapitre 1

Connaissances nécessaires au système expert

Chapitre 2

Implémentation du système expert

2.1 Choix de la représentation Lisp

2.1.1 Base de règles

Pour représenter les règles de notre SE, nous avons choisi d'utiliser une liste qui contiendra 2 sous-listes (les nouveaux faits et les prémisses) ainsi que le nom de la règle. Par exemple, voici la règle RCR1 qui permet de savoir si un chantier est réalisable :

```
(
        (  ;;;;  DEBUT DES NOUVEAUX FAITS
                (ChantierRealisable . T)
                (EquipeDeNuit . NIL)
        );;;;  FIN DES NOUVEAUX FAITS
                (;;;;DEBUT DES PREMISSES
                (COMPARAISON (>=
                        (* LargeurTunnel HauteurTunnel
                            NombreDeJours VitesseNain)
                        LongueurTunnel
                ))
        );;;;FIN DES PREMISSES
RCR1);;;;NOM DE LA REGLE
```

On peut noter que les premisses ont un format bien spécial. En effet, chaque condition commence par le type de vérification à faire. Cela peut être une égalité, une comparaison, ou encore une définition (savoir si un fait a déjà été défini). Cette étiquette permet de simplifier le traitement lors de la vérification des prémisses. Pour les prémisses, une fonction va donc vérifier si LargeurTunnel * HauteurTunnel * NombreDeJours * VitesseNain est bien supérieur ou égale à la longueur du tunnel. Si c'est le cas, notre moteur d'inférence va apeller une nouvelle fonction qui va évaluer les valeurs des nouveaux faits et les ajouter à notre base de fait. Nous avons également une A-List qui contient la description

de chaque règles, mais celle-ci a été mise à part dans le but de garder un code clair. Cette représentation permet non seulement d'avoir un format homogène pour toutes les règles, et donc de traiter chaque règle de la même façon, mais elle permet également de traiter des conditions complexes qui demandent l'évaluation de plusieurs opérations ou encore des opérations qui demandent de rechercher dans la base de faits.

2.1.2 Base de faits

La base de faits a été faites de la manière la plus simple possible, celle-ci est juste une liste donc chaque membre est une sous-liste qui représente un fait. Un fait est tout simplement constitué d'une étiquette, donc le nom du fait, et d'une valeur qui peut être un nombre, un symbole, ou même une valeur booléenne.

```
(
  (VitesseNain 3)
  (TypeRoche GRANITE)
  (LargeurTunnel 3)
  (LongueurTunnel 28)
  (HauteurTunnel 1)
)
```

Ici on peut voir qu'il y a 4 faits dans notre base de faits : la vitesse des nains, le type de roche, la largeur et la longueur du tunnel, et la hauteur du tunnel.

2.2 Le moteur d'inférences

Le moteur d'inférence est un moteur à chaînage avant avec un fonctionnement simple, mais pour bien comprendre comment il fonctionne, il faut expliciter certaines fonctions qui permettent de savoir si les prémisses sont satisfaits ou bien d'ajouter les faits à notre base de faits. Nous allons donc commencer par présenter toutes ces fonctions.

2.2.1 Validation des prémisses

Pour que les prémisses soient validés, il faut que toutes les conditions soient validées. Donc nous allons itérer sur toutes les conditions de la liste des prémisses pour vérifier qu'elles sont toutes validée. Mais certaines conditions sont complexes et demandent d'aller vérifier l'existence ou la valeur de faits dans la base de faits. Pour cela nous avons défini une fonction evalOperande qui va nous permettre d'évaluer la valeur de chaque opérande :

```
(defun evalOperande(operande)
  (let ((newList (list)))
      (dolist (x operande newList)
        (cond
          ((LISTP x)
            (setq newList (append newList (list (evalOperande x)))))
          ((OR (EQUAL x '*) (EQUAL x '/) (EQUAL x '-) (EQUAL x '+))
            (setq newList (APPEND newList (list x))))
          ((SYMBOLP x)
            (if (NULL (ASSOC x *BaseFaits*))
              ;;;; SI LE SYMBOLE N'EST PAS DEFINI DANS LA BDF
                (return-from evalOperande NIL)
              (setq newList (APPEND newList (cdr (ASSOC x *BaseFaits*))))
            ))
          (T (setq newList (APPEND newList (list x))))
        )
    )
  )
)
```

2.2.2 Calcul des nouvelles valeurs

2.2.3 Moteur à chaînage avant

2.2.4 Test du moteur d'inférences

Chapitre 3

Scénarios d'utilisation

Conclusion