

# Cours IA ESP-UCAD

Pr. Mamadou Camara  
mamadou.camara@esp.sn

2021-2022

# Table des matières

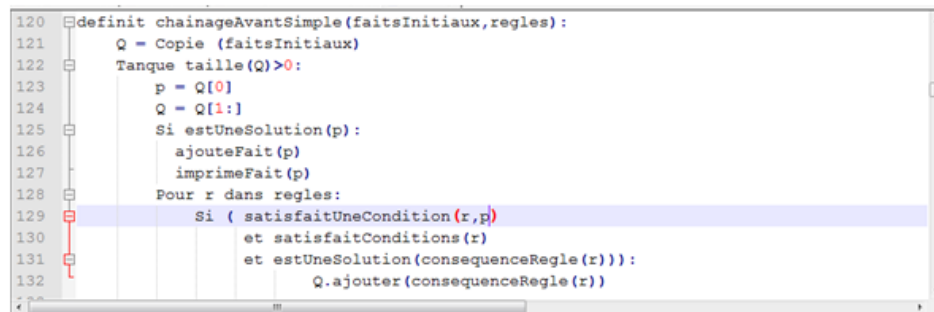
<b>1</b>	<b>TP Chaînage avant simple</b>	<b>2</b>
1.1	TP : Application du Chaînage avant . . . . .	2
1.2	Cas d'étude : Un système expert bancaire [Projet Master GI] . . . . .	2
1.3	Cas d'étude : ville méritant le voyage [DIC-DIT Info] . . . . .	3
1.4	TP : Chaînage avant simple . . . . .	4
1.4.1	Définition de faits et de règles . . . . .	4
1.4.2	Affichage d'un fait . . . . .	6
1.4.3	Accès à la conséquence et aux conditions d'une règle . . . . .	6
1.4.4	Vérification de la présence d'un fait dans une règle . . . . .	7
1.4.5	Vérification qu'une règle peut être déclenchée . . . . .	8
1.4.6	Le prédicat estUneSolution . . . . .	8
1.4.7	Chaînage avant . . . . .	9
1.5	Récupération des données par entrée clavier . . . . .	10

# Chapitre 1

## TP Chaînage avant simple

### 1.1 TP : Application du Chaînage avant

Écrire le code lisp correspondant à l'algorithme chaînage avant simple.



Répondre aux questions ci-après en considérant l'exemple Cs\_Bulding décrit dans l'exercice 1.

1. Traduire chaque prémisse de l'exemple en logique des prédicats d'ordre zéro
2. Illustrer le fonctionnement de l'algorithme chaînage avant simple en utilisant les données de l'exemple.
  - utiliser les numéros des lignes pour identifier les instructions

### 1.2 Cas d'étude : Un système expert bancaire [Projet Master GI]

Une banque utilise un système expert pour accorder un prêt. Les variables suivantes sont employés pour décrire les propositions associées :

- OK : le prêt est accordé
- CO : le conjoint se porte garant
- PA : le candidat au prêt peut payer ses traites
- RE : le dossier du candidat est bon
- AP : les revenus du conjoint sont élevés
- RA : le taux d'intérêt est faible
- IN : les revenus du candidat sont supérieurs à ses dépenses
- BA : le candidat n'a jamais de découvert sur son compte courant
- MB : le conjoint doit hériter

Les règles sont les suivantes

1. Si MB Alors CO
2. Si AP Alors CO
3. Si RA Alors RE

4. Si IN Alors PA
5. Si BA,RE Alors OK
6. Si CO,PA,RE Alors OK

Considérons la base de faits initiale BA, RA, MB, AP, IN avec OK pour but à établir. Il est demandé d'utiliser le moteur d'inférence que vous venez de construire pour réaliser l'inférence sur le client décrit par cette base de faits.

Utiliser le système d'initialisation de la BF par chargement de fichier en utilisant plusieurs cas de test. Tester toutes les fonctions définies dans la section Chainage avant simple en utilisant les données "système expert bancaire".

### 1.3 Cas d'étude : ville méritant le voyage [DIC-DIT Info]

Nous voulons créer un système qui assiste l'utilisateur à décider si la ville qu'il s'apprête à visiter, mérite d'être visitée (ville méritant le voyage). Considérons la base de règles suivante :

1. Soit la base de règles suivante :
2. si ville historique alors ville méritant le voyage
3. si ville artistique alors ville méritant le voyage
4. si nombreuses animations alors ville méritant le voyage
5. si ville agréable et tradition gastronomique alors ville méritant le voyage
6. si belle ville et nombreux monuments alors ville artistique
7. si ville ancienne et nombreux monuments alors ville historique
8. si nombreux concerts et nombreux théâtres alors nombreuses animations
9. si activités sportives et traditions folkloriques alors nombreuses animations
10. si espaces verts et climat agréable alors ville agréable
11. si espaces verts et nombreux monuments alors belle ville
12. si nombreux restaurants et bons restaurants alors tradition gastronomique

Choisir un ensemble de faits parmi ceux listés ci-après de sorte à pouvoir lancer le moteur et trouver le but "ville méritant le voyage" :

1. nombreux monuments,
2. ville ancienne,
3. nombreux concerts,
4. nombreux théâtres,
5. activités sportives,
6. traditions folkloriques,
7. espaces verts,
8. climat agréable,
9. nombreux restaurants,
10. bons restaurants.

Utiliser le système d'initialisation de la BF par chargement de fichier en utilisant plusieurs cas de test. Tester toutes les fonctions définies dans la section Chainage avant simple en utilisant les données "ville méritant le voyage".

## 1.4 TP : Chaînage avant simple

Nous allons réaliser un moteur d'inférence à chaînage avant simple sans variables, s'appliquant à des règles sans variables. Le langage utilisé sera Lisp. Comme nous l'avons décrit dans le chapitre 3, l'idée de base est de déduire tout ce qu'il est possible à partir d'un ensemble de faits initiaux et d'un ensemble de règles. Les règles que nous considérerons seront des clauses de Horn. Elles sont composées de deux parties :

- Un ensemble de conditions qui doivent toutes être satisfaites pour que la règle se déclenche.
- Une seule conséquence, i.e., un nouveau fait, qui devra être inséré dans la base des faits.

A chaque fois qu'un nouveau fait est déduit, l'ensemble complet des règles doit être appliqué à nouveau à la base des faits : le nouveau fait peut permettre le déclenchement d'une règle qui avait déjà été essayée sans succès. Le processus d'inférence se termine lorsque plus aucun nouveau fait ne peut être déduit. Pour tester les fonctions, vous pouvez définir deux fichiers :

1. Un fichier `tp1` qui contient les fonctions à définir
2. Un fichier de test dans lequel sera appelé le fichier `tp1`. La capture ci-après montre les premières lignes de ce fichier de test.

```
1 ;commentaire sur une ligne
2
3
4
5 ; (load "C:/Lecture/M1/IA/Lisp/Material/tp1/testertpbq.lsp")
6 (defmacro while (test &rest body)
7   "Repeat body while test is true."
8   (list* 'loop
9         (list 'unless test '(return nil))
10        body))
11
12 (load "C:/Lecture/M1/IA/Lisp/Lisp/tp/tp1/tp1.lisp")
13
```

### 1.4.1 Définition de faits et de règles

Les faits et les règles seront stockés dans les bases de données de portée globale faits, respectivement règles. Celles-ci sont codées comme des listes initialement vides

```
1 (defvar echec 'echec)
2 (defvar faits nil)
3 (defvar regles nil)
4
```

Un fait sera un atome ('pas-d-enfants' par exemple) tandis qu'une règle sera une liste (`< condition1 > ... < conditionn > < consequence >`). Les faits ainsi que les règles seront stockés tels quels dans les bases de données. Les fonctions `ajouteFait` et `ajouteRegle` vous permettant d'ajouter des faits et des règles aux bases de données concernées. `ajouteFait` n'accepte qu'un seul paramètre, le fait (un atome), tandis que `ajouteRegle` a deux paramètres : la liste des conditions de la règle et la conclusion (un atome) de la règle. La règle elle-même est enregistrée comme une liste dont le premier élément est la liste des conditions et la seconde représente la conclusion.

```

15  (defun ajouteFait (fait)
16  .....
17
18  )
19
20  (defun ajouteRegle(conditions consequence)
21  .....
22
23  )

```

La fonction `initDBs` permet de réinitialiser les bases de données des règles `regles` et des faits `faits`. Grâce à ces trois fonctions `ajouteFait`, `ajouteRegle` et `initDBs`, il est maintenant possible d'enregistrer des règles et des faits que l'on peut directement évaluer dans l'interpréteur. Voici quelques exemples qui vont vous permettre de tester vos fonctions. Tout d'abord, nous ajoutons quelques faits à l'aide de la fonction `ajouteFait`.

```

54  (defun remplir_faits1 ()
55  (ajouteFait 'jamais-de-decouvert)
56  (ajouteFait 'taux-interet-faible)
57  (ajouteFait 'conjoint-doit-heriter)
58  (ajouteFait 'revenus-conjoint-eleves)
59  (ajouteFait 'revenus-superieurs-depenses)
60  )

```

Ensuite, nous définissons les règles suivantes avec `ajouteRegle`.

```

16  (defun remplir_regles()
17  (ajouteRegle '(conjoint-doit-heriter) 'conjoint-garant)
18  (ajouteRegle '(revenus-conjoint-eleves) 'conjoint-garant)
19

```

Les faits les règles sont chargés après avoir réinitialisé les deux bases à l'aide de la fonction `initDBs`.

```

77  (defun remplir_1()
78  (initDBs)
79  (remplir_faits1)
80  (remplir_regles)
81  )
82  (remplir_1)
83

```

```

GNU CLISP 2.49
"
nous allons afficher les faits
"
<REVENUS-SUPERIEURS-DEPENSES REVENUS-CONJOINT-ELEVES CONJOINT-DOIT-HERITER
TAUX-INTERET-FAIBLE JAMAIS-DE-DECOUVERT>
"
nous allons afficher les regles
"
<<<CONJOINT-GARANT PEUT-PAYER-TRAITES DOSSIER-BON> PRET-ACCORDE>
<<JAMAIS-DE-DECOUVERT DOSSIER-BON> PRET-ACCORDE>
<<REVENUS-SUPERIEURS-DEPENSES> PEUT-PAYER-TRAITES>
<<TAUX-INTERET-FAIBLE> DOSSIER-BON>
<<REVENUS-CONJOINT-ELEVES> CONJOINT-GARANT>
<<CONJOINT-DOIT-HERITER> CONJOINT-GARANT>>
"

```

### 1.4.2 Affichage d'un fait

Pour afficher les nouveaux faits déduits au fur et à mesure qu'ils sont ajoutés à la base des faits, nous écrivons la fonction `imprimeFact` qui permet d'afficher un fait passée en paramètre.

```

125
126 (print "Affichons le premier fait")
127 (imprimeFact (nth 0 faits))

```

### 1.4.3 Accès à la conséquence et aux conditions d'une règle

Pour pouvoir accéder aux conditions et à la conséquence d'une règle, nous écrivons deux fonctions. `conditionsRegle` retourne la liste des conditions de la règle passée en paramètre. `consquenceRegle` retourne la conséquence de la règle passée en paramètre.

```

127 "
128 nous allons afficher consequenceRegle de la regle 0
129 "
130 (print (consquenceRegle (nth 0 regles)))
131
132 (print "
133 nous allons afficher conditionsRegle de la regle 0
134 ")
135 (print (conditionsRegle (nth 0 regles)))
136

```

```

GNU CLISP 2.49
"
nous allons afficher consequenceRegle de la regle 0
"
PRET-ACCORDE
"
nous allons afficher conditionsRegle de la regle 0
"
<CONJOINT-GARANT PEUT-PAYER-TRAITES DOSSIER-BON>

```

### 1.4.4 Vérification de la présence d'un fait dans une règle

La fonction `satisfaitUneCondition` vérifie si un fait `q` fait parti des conditions d'une règle. Elle prend deux arguments en paramètre : un fait et une règle, et retourne `True` si un fait passé en paramètre est égale à une des conditions de la règle, sinon `False`.

```
GNU CLISP 2.49
"
nous allons afficher les faits
"
<REVENUS-SUPERIEURS-DEPENSES REVENUS-CONJOINT-ELEVES CONJOINT-DOIT-HERITER
TAUX-INTERET-FAIBLE JAMAIS-DE-DECOUVERT>
"
nous allons afficher les regles
"
<<CONJOINT-GARANT PEUT-PAYER-TRAITES DOSSIER-BON> PRET-ACCORDE>
<<JAMAIS-DE-DECOUVERT DOSSIER-BON> PRET-ACCORDE>
<<REVENUS-SUPERIEURS-DEPENSES> PEUT-PAYER-TRAITES>
<<TAUX-INTERET-FAIBLE> DOSSIER-BON>
<<REVENUS-CONJOINT-ELEVES> CONJOINT-GARANT>
<<CONJOINT-DOIT-HERITER> CONJOINT-GARANT>
"
nous allons afficher la taille de la base de regles
"
6
```

```
137 (print "regle 2 et fait 0")
138 (print (nth 2 regles))
139 (print (nth 0 faits))
140 (print "
141 Verifion satisfaitUneCondition regle 2 et fait 0
142 ")
143
144
145 (print (satisfaitUneCondition (nth 2 regles) (nth 0 faits)))
146 ;-----
```

```
GNU CLISP 2.49
"regle 2 et fait 0"
<<REVENUS-SUPERIEURS-DEPENSES> PEUT-PAYER-TRAITES>
REVENUS-SUPERIEURS-DEPENSES
"
Verifion satisfaitUneCondition regle 2 et fait 0
"
T
"
```

```
GNU CLISP 2.49
"regle 1 et fait 4"
<<JAMAIS-DE-DECOUVERT DOSSIER-BON> PRET-ACCORDE>
JAMAIS-DE-DECOUVERT
"
Verifion satisfaitUneCondition regle 1 et fait 4
"
T
```



```

GNU CLISP 2.49
T
"regle 1 et fait 0"
<<JAMAIS-DE-DECOUVERT DOSSIER-BON> PRET-ACCORDE>
REVENUS-SUPERIEURS-DEPENSES
"
Verifion satisfaitUneCondition regle 1 et fait 0
"
NIL

```

### 1.4.5 Vérification qu'une règle peut être déclenchée

Vous devez maintenant vérifier si une règle peut être déclenchée. Définissez la fonction `satisfaitConditions`, qui prend en paramètres une règle. Elle retournera `True` si toutes les conditions de la règle peuvent être satisfaites par les faits de la base de données. `False` sinon. Cette fonction **utilise une fonction inclusion** qui permet de tester si les éléments d'une liste sont tous présents dans une autre liste.

```

170
171 (print "
172 nous allons afficher satisfaitConditions de la regle 2
173 ")
174 (print (satisfaitConditions(nth 2 regles)))
175 ;-----
176 (print "
177 nous allons afficher satisfaitConditions de la regle 1
178 ")
179 (print (satisfaitConditions(nth 1 regles)))

```

```

GNU CLISP 2.49
nous allons afficher satisfaitConditions de la regle 2
"
T
"
nous allons afficher satisfaitConditions de la regle 1
"
NIL

```

### 1.4.6 Le prédicat `estUneSolution`

La fonction `estUneSolution` vérifie si un fait est une solution. Une solution est un fait qui ne se trouve pas dans la base des faits. La fonction retourne `True` si le fait est une solution ; sinon `False`

- `defun estUneSolution(fait) : Il est nécessaire de définir une façon récursive un prédicat "defun estdans(i l)" qui vérifie si un éléments i est dans une liste l. Ce prédicat sera appelé par estUneSolution.`

```

184 (print (nth 3 regles))
185
186 (print "
187 nous allons afficher estUneSolution de consequenceRegle de la regle 3
188 ")
189
190
191 (print (estUneSolution(consequenceRegle (nth 3 regles))))

```

```

GNU CLISP 2.49

"regle 3"
<<TAUX-INTERET-FAIBLE> DOSSIER-BON>
"
nous allons afficher estUneSolution de consequenceRegle de la regle 3
"
T
"

```

### 1.4.7 Chaînage avant

La fonction principale `chainageAvantSimple`, prend en entrée la liste des règles et la liste des faits initiaux, et affiche à l'écran les nouveaux faits déduits. Comme nous n'avons pas de variables dans cette première version, aucun filtrage n'est nécessaire. Les faits initiaux et inférés sont gérés par la file d'attente : `Q` (qui contient au début les faits initiaux). S'il n'y est pas déjà, chaque fait de `Q` va être ajouté dans la base des faits et toutes les règles vont être essayées pour tenter d'inférer de nouveaux faits, **qui seront ajoutés en queue de `Q`**. Le processus s'arrête dès que `Q` est vide.

```

120 defint chainageAvantSimple(faitsInitiaux,regles):
121   Q = Copie (faitsInitiaux)
122   Tantque taille(Q)>0:
123     p = Q[0]
124     Q = Q[1:]
125     Si estUneSolution(p):
126       ajouteFait(p)
127       imprimeFait(p)
128     Pour r dans regles:
129       Si ( satisfaitUneCondition(r,p)
130          et satisfaitConditions(r)
131          et estUneSolution(consequenceRegle(r))):
132         Q.ajouter(consequenceRegle(r))

```

Lancer la fonction chaînage avant simple et afficher la nouvelle base de faits.

```

248 ;-----
249 (print "
250 nous allons lancer le chainageavantsimple
251 ")
252 (chainageAvantSimple faits regles)
253 ;-----
254 (print "
255 nous allons afficher les faits
256 ")
257 (print faits)
258

```

```

GNU CLISP 2.49

T
"
nous allons lancer le chainageavantsimple
"
PEUT-PAYER-TRAITES
CONJOINT-GARANT
DOSSIER-BON
PRET-ACCORDE
"
nous allons afficher les faits
"
<PRET-ACCORDE DOSSIER-BON CONJOINT-GARANT PEUT-PAYER-TRAITES
REVENUS-SUPERIEURS-DEPENSES REVENUS-CONJOINT-ELEVES CONJOINT-DOIT-HERITER
TAUX-INTERET-FAIBLE JAMAIS-DE-DECOUVERT>
;; Loaded file C:\Lecture\M1\IA\Lisp\Material\tp1\testertpbq.lsp
T
Break 1 [51]>

```

## 1.5 Récupération des données par entrée clavier

Il est demandé de modifier le code Tp1 afin de permettre à l'utilisateur d'initialiser la base de fait en répondant à une série de questions par entrée clavier. Vous pouvez définir les fonctions suivantes :

1. **ajouterquestion** permet d'insérer une nouvelle question dans la base de questions
2. **remplirquestions** permet de remplir la base de questions avec plusieurs appels de la fonction ajouterquestion
3. **poserquestion** permet d'afficher une question et ses options de réponse et d'ajouter le fait correspondant au choix tapé par l'utilisateur (utiliser la fonction read).
4. **posertoutesquestion** : permet de parcourir la base de question et d'appeler poserquestion pour chaque question