Rapport de projet LO21 / Système expert

Diebolt Matthieu

Pierson Baptiste

Table des matières

I. Choix de conception et d'implémentation

1. Description des choix relatifs aux structures de données

Notre système expert est composé de 3 structures de donnés différentes : les règles, la base de connaissances et la base de faits.

a) Les règles

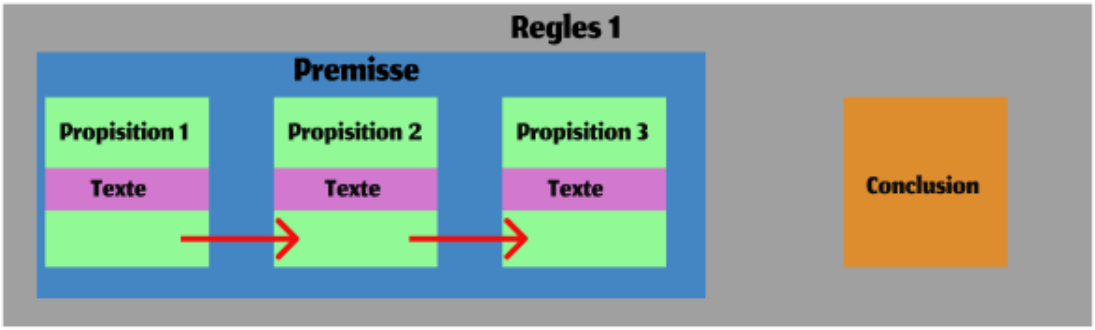
Les règles sont composées de 2 parties, une prémisse pouvant contenir plusieurs propositions et une conclusion unique.

Ainsi en c, nous avons défini les règles comme une structure de données abstraites composée d'une prémisse et d'une conclusion.
Nous avons défini la conclusion de type char et la prémisse comme une liste chainée contenant l'élément suivant de la proposition et son texte.

Voici notre déclaration :

```
typedef struct proposition {
        char *texte;
        struct proposition *next;
}Proposition;

typedef Proposition *Premisse;

typedef char *Conclusion;

typedef struct Regle {
        Premisse premisse;
        Conclusion conclusion;
}regle;
```

Voici un schéma de la structure :



b) La base de connaissances

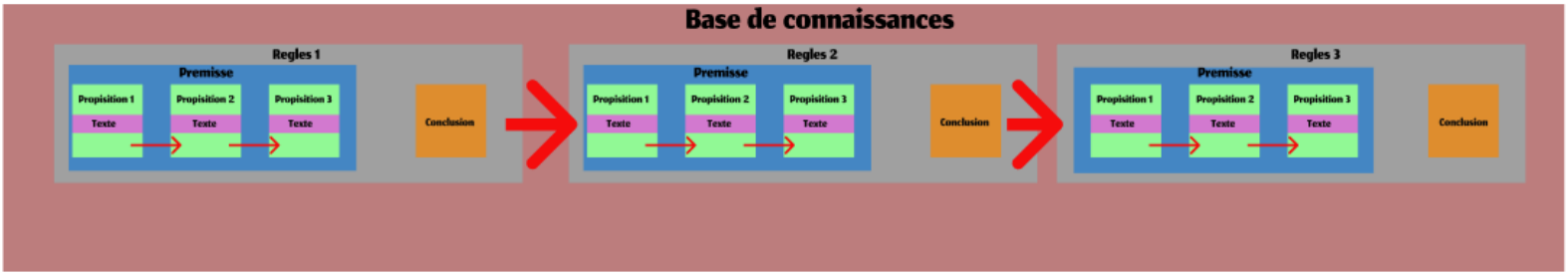
La base de connaissances est composée de plusieurs règles.

En c, nous avons défini cette structure comme une liste chainée contenant une règle et pointant sur la prochaine règle.

Voici notre déclaration :

```
typedef struct baseC{
  regle  Bcregle;
  struct baseC *next;
} baseConnaissance;

typedef baseConnaissance *BC;
```

Voici un schéma de la structure :



c) La base de faits

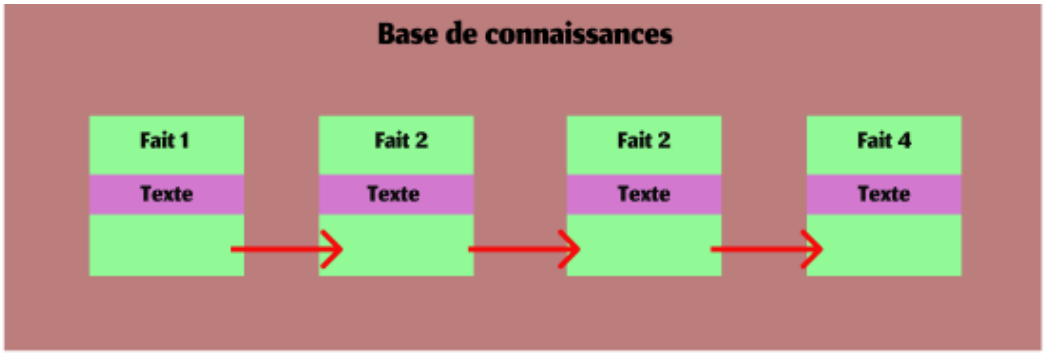
La base de faits est composée de plusieurs propositions considérées comme vraies par l'utilisateur. (Elle est similaire à la prémisse d'une règle).

Ainsi en c, nous avons déclaré cette structure de façon similaire à la prémisse. Il s'agit donc d'une liste chainée.

Voici notre déclaration :

```
typedef struct baseF {
        char *texte;
        struct baseF *next;
}baseFaits;

typedef baseFaits *BF;
```

Voici un schéma de la structure :



II. Algorithmes

Sur nos structures de données, nous appliquons des opérations. Voici la plupart de leur algorithme :

1. Opérations sur les règles

a) Créer une règle vide

Profil :

Données :
Résultat : règle (vide)
Lexique :

- L = règle
- Créer() = créer une liste chainée
- Creerpremisse() = créer la premisse vide d'une règle
- Créerconclusion() = créer la conclusion vide d'une règle

Algorithme :

```
CréerRegle( ) → regle
{
      l=créer()
      CréerPremisse(l)
      CréerConclusion(l)
      CréerRegle()=l
}
```

b) Ajouter une proposition à la prémisse d'une règle, l'ajout se fait en queue

Profil :

Données : règle, texte
Résultat : règle
Lexique :

- reg = structure abstraite composée d'une prémisse et d'une conclusion
- Prop = une proposition définie par l'utilisateur (du texte)
- Actuel = proposition courante d'une prémisse
- Estvidepremisse = renvoie vrai si la prémisse d'une règle ne contient aucune proposition
- ajouterP = opération qui ajoute une proposition à la suite d'un élément dans une liste chainée
- AjoutTête=ajoute une proposition dans une liste Vide
- Suivant() = définit le suivant d'un élément de la liste chainée

Algorithme :

```
Ajtpremisse(reg :regle , prop : texte)->regle
{
      Si  estvidepremisse(reg) alors
            ajouterTête(reg, prop)
      Sinon
            Actuel=tete(premisse(reg))
            Tant que suivant(actuel) ≠indefini faire

                  actuel = suivant(actuel)
            fait
            ajouterP(actuel, prop)

      Fsi
      Ajtpremisse(reg,prop)= reg
}
```

c) Créer la conclusion d'une règle

Profil :

Données : regle, txtconclusion
Résultat : regle
Lexique :

- reg = structure abstraite composée d'une prémisse et d'une conclusion
- txtconc = une conclusion définie par l'utilisateur (du texte)
- Estvidepremisse = renvoie vrai si la prémisse d'une règle ne contient aucune proposition
- AjouterConc() = définit la conclusion d'une règle

Algorithme :

```
Creerconclusion(reg :regle,txtconc:texte)-> regle
{
      si non Estvidepremisse(reg) alors
            reg=ajouterConc(reg,textconc)
      fsi

      Creerconclusion(reg, txtconc) = reg
}
```

d) Tester si une proposition appartient à la prémisse d'une règle, de manière récursive

Profil :

Données : prémisse, proposition
Résultat : booléen
Lexique :

- prem = structure de données abstraites contenant des propositions
- prop = une proposition définie par l'utilisateur (du texte)
- Estvidepremisse = renvoie vrai si la prémisse d'une règle ne contient aucune proposition
- Tete() = donne l'élément en tête d'une liste chainée
- Reste() = donne le reste de la liste chainée sans la tête

Algorithme :

```
apartPremisse(prem : premisse, prop : texte)-> booleen
{
      Si estvidepremisse(prem) alors
            apartPremisse(prem, prop) = faux
      sinon
            si proposition(val(tete(prem)))=prop alors
                  apartPremisse(prem, prop) = vrai
            sinon
                  apartPremisse(prem, prop) =
                  apartPremisse(reste(prem), prop)
            fsi
      fsi
}
```

e) Supprimer une proposition de la prémisse d'une règle

Profil :

Données : règle, proposition
Résultat : règle
Lexique :

- reg = structure abstraite composée d'une prémisse et d'une conclusion
- Prop = une proposition définie par l'utilisateur (du texte)
- Actuel = proposition courante d'une prémisse
- Estvidepremisse = renvoie vrai si la prémisse d'une règle ne contient aucune proposition
- apartPremisse() = Teste si une proposition appartient à la prémisse d'une règle
- tete() = donne l'élément en tête d'une liste chainée
- premisse() = donne la liste chainée contenant les propositions d'une règle
- suivant() = donne l'élément suivant de la liste chainée

- supprop() = supprime l'élément entre 2 éléments donnés
- reste() = renvoie le reste de la liste chainée sans la tête

Algorithme :

```
supProposition(reg :regle, prop : texte)->regle

{
      Si estvidePremisse(reg) alors
                  supProposition(reg , prop) = reg
      sinon si apartPremisse(reg, prop) = VRAI alors
                  si proposition(val(tete(premisse(reg))) = prop alors
                        premisse(reg) = reste(premisse(reg)
                        supProposition(reg , prop) = reg
                  sinon
                        actuel = premisse(reg)
                        tant que Proposition(val(suivant(actuel))) ≠ prop
                        faire
                              actuel = suivant(actuel)
                        fait
                        supprop(actuel,suivant(suivant(actuel))
                        supProposition(reg,prop) = reg
                  fsi
            sinon supProposition(reg , prop) = reg
            fsi
      fsi
}
```

f) Tester si la prémisse d'une règle est vide

Profil :

Données : prémisse
Résultat : booleen
Lexique :

- Prem = structure de données abstraites contenant des propositions
- tete() = donne l'élément en tête d'une liste chainée

Algorithme :

```
Estvidepremisse( prem :premisse)->booleen
{
Si tete(prem) = indefini alors
      Estvidepremisse(prem) = vrai
Sinon
      Estvidepremisse(prem) = faux
Fsi
}
```

g) Accéder à la proposition se trouvant en tête d'une prémisse

Profil :

Données : premisse
Résultat : proposition
Lexique :

- prem = structure de données abstraites contenant des propositions
- Estvidepremisse = renvoie vrai si la prémisse d'une règle ne contient aucune proposition
- tete() = donne la valeur de l'élément en tête d'une liste chainée

Algorithme :

```
Tetepremise(prem : premisse)->proposition
{
   Si estvidePremisse(prem) alors
      Tetepremise(prem) = Indefini
   Sinon
      Tetepremisse(prem) = proposition(val(tete(prem)))
fsi
}
```

h) Accéder à la conclusion d'une règle

Profil :

Données : regle
Résultat : conclusion
Lexique :

- reg = structure abstraite composée d'une prémisse et d'une conclusion
- estvideConclussion() = retourne vrai si la conclusion d'une règle est vide

Algorithme :

```
conclRegle(reg : regle)->conclusion
{
      si estvideConclusion(reg) alors
            conclRegle(reg) = indefini
      sinon
            conclRegle(reg)  = conclusion(val(tete(reg)))
      fsi
}
```

2. Operations sur les bases de connaissance

a) Créer une base vide

Profil :

Données : X
Résultat : base de connaissance
Lexique :

- baseC = structure abstraite composée de règles
- Créer() = créer une liste chainée

Algorithme :

```
creerbaseC()->base de connaissance
{
      baseC = créer()
      creerbaseC() = baseC
}
```

b) Ajouter une règle à une base, l'ajout peut se faire en queue

Profil :

Données : baseC , règle
Résultat : base de connaissance
Lexique :

- baseC = base de connaissance = structure abstraite composée de règles
- reg = structure abstraite composée d'une prémisse et d'une conclusion
- Estvidebc = renvoie vrai si la base de connaissance est vide, faux sinon
- Suivant() = définit le suivant d'un élément de la liste chainée
- ajouterBC = opération qui ajoute une règle à la suite d'un élément dans une liste chainée
- AjoutTête=ajoute une règle dans une liste Vide

Algorithme :

```
AjouterRegleC(baseC : base de connaissance, reg : regle)->baseC
{
      Si  estvidebc(baseC) alors
            ajouterTête(baseC, reg)
      Sinon
            actuel=tete(regle(baseC))
            Tant que suivant(actuel) ≠indefini faire

                  actuel = suivant(actuel)
            fait
            ajouterBC(actuel, reg)
      Fsi
      AjouterRegleC (baseC, reg)= baseC
}
```

c) Accéder à la règle se trouvant en tête de la base

Profil :

Données : base de connaissances
Résultat : regle
Lexique :

- baseC = structure abstraite composée de règles
- regle = structure abstraite composée d'une prémisse et d'une conclusion
- tete() = donne la valeur de l'élément en tête d'une liste chainée

Algorithme :

```
tetebaseC(baseC : base de connaissance)->regle
{
      tetebaseC(baseC) = tete(baseC)
}
```

3. Moteur d'inférence

Profil :

Données : Base de connaissance, Base de faits
Résultat : X
Lexique :

- baseC = structure abstraite composée de règles
- baseF = structure abstraite composée de propositions considérées comme vraie par l'utilisateur.
- tete() = donne l'élément en tête d'une liste chainée
- Regleactuelle = variable pour stocker la règle dans laquelle on est en train de chercher
- concTrouver = variable pour savoir si on a trouvé au moins une conclusion
- premisseactuelle = variable pour stocker la prémisse dans laquelle on est en train de chercher
- premisse() = donne la liste des propositions d'une regle
- apartpremisseF () = fonction qui teste si tous les éléments de la prémisse sont présents dans la base de faits, renvoie vrai dans ce cas faux sinon
- premisseTrouver = variable pour savoir si toutes les prémisses ont un équivalent dans la base de faits ou non
- suivant() = donne l'élément suivant d'une liste
- afficher() = permet d'afficher un message pour l'utilisateur
- ajouterfait() = permet d'ajouter un fait à la base de fait

Description :

Parcourt les règles de base de connaissances et pour chacune vérifie si toutes les propositions de la prémisse se trouve dans la base de fait. Si c'est le cas, la conclusion est rajoutée dans la base de faits et est affichée

Algorithme

```
moteurInf(baseC : base de connaissance, baseF : base de faits)
{

Regleactuelle = tete(baseC)
concTrouver = 0

tant que Regleactuelle ≠ Indefinie faire
      premisseactuelle = tete(premisse(regleactuelle)
      premisseTrouver = 1
      si non apartpremisseF(premisseActuelle, baseF) alors
                premisseTrouver = 0
      fsi
      si premisseTrouver = 1 faire
            afficher(conclusion(regleactuelle)
            ajouterfait(conclusion(regleactuelle))
            concTrouver = 1
      fsi
      regleactuelle = suivant(regleactuelle)
fait

si concTrouver = 0 alors
      afficher(« aucune conclusion trouvée »)
fsi


}
```

apartpremisseF ()

Description :

Parcourt la liste des faits pour voir si toutes les propositions de la prémisse existent dans la base de faits

Profil :

Données : premisse, Base de faits
Résultat : Bollene
Lexique :

- premisse = structure abstraite composée de propositions
- baseF = structure abstraite composée de propositions considérées comme vraies par l'utilisateur.

- Estvidepremisse = renvoie vrai si la prémisse est vide
- faitactuel = variable pour stocker le fait actuel avec lequel se fait la comparaison
- tete() = donne l'élément en tête d'une liste chainée
- suivant() = donne l'élément suivant d'une liste

Algorithme :

```
apartpremisseF(prem: premisse, baseF : base de faits)
{
 Si estvidepremisse(prem) alors
     apartpremisseF(prem, baseF) =  vrai
  sinon
          faitactuel = tete(baseF)
          tant que faitactuel ≠ Indefinie faire
               si tete(prem) = tete(faitactuel) alors
                    apartpremisseF(prem, baseF ) =
                    apartpremisseF(suivant(prem), baseF )
               fsi
               faitactuel = suivant(faitactuel)
          fait
          apartpremisseF(prem, baseF) = faux
fsi

}
```

III. Jeux d'essais

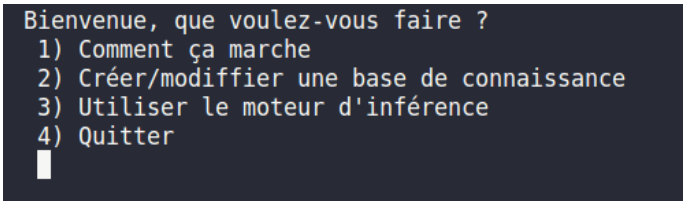
Nous avons voulu créer un programme qui offre beaucoup de possibilités à l'utilisateur. Ainsi c'est lui qui définit ses règles, ses bases de connaissances, ses bases de faits. Cela lui donne la possibilité d'utiliser le programme dans différents domaines.

La base de connaissances est enregistrée dans un fichier que l'utilisateur définit lui-même. Dans ce fichier sont enregistrées toutes les règles.

L'utilisateur peut ainsi créer autant de bases qu'il le souhaite, chacune étant stockée dans un fichier différent. Les différentes bases sont conservées et pourront être modifiées ultérieurement ou supprimées. Tout cela est possible grâce aux listes de choix proposés.

Nous guidons l'utilisateur dans l'élaboration de sa base de connaissances.

Une fois qu'il lance le programme le programme, une liste de choix est proposée.



- Il peut obtenir de l'aide (choix1) :

- Il peut créer ou modifier sa base de connaissance (choix 2). Il peut alors :
  - Soit créer une nouvelle base de connaissances (choix 2)

```
matthieu@matthieu-VirtualBox:~/Bureau/Lo21$ ./main
Bienvenue, que voulez-vous faire ?
 1) Comment ça marche
 2) Créer/modiffier une base de connaissance
 3) Utiliser le moteur d'inférence
 4) Quitter
 2

------------------
Que voulez vous faire ?
 1) Modifier une base deja existante ?
 2) Creer une nouvelle base de connaissance ?
 3) Quitter
2

Quelles nom voulez vous donner a la base ?

essai1
La base a ete creer

Veuillez saisir 1 pour ecrire une nouvelle regle ou 2 pour quitter:
 1

 taper 1 pour ajouter une premise et 2 pour donner la conclusion
1


 Donner premisse à la regle :

prem1

 prem1 a bien été ajouter


 taper 1 pour ajouter une premise et 2 pour donner la conclusion
1


 Donner premisse à la regle :

prem2

 prem2 a bien été ajouter


 taper 1 pour ajouter une premise et 2 pour donner la conclusion
2
```

```
 Donner la conclusion à la regle :

concl1

 concl1 a bien été ajouter
------------------------
 Voici votre regle ajouter:

 Premisse:
- prem1
- prem2
Conclusion:
 concl1
------------------------

 voici votre base actuelle :

 Prémisse: prem1 prem2
Conclusion: concl1


Veuillez saisir 1 pour ecrire une nouvelle regle ou 2 pour quitter:
```

Il devra alors donner le nom de la base à créer. Le nom qu'il donnera correspondra au nom du fichier qui contient les règles.
Ensuite, il pourra créer une règle. Pour cela , il pourra définir une prémisse en ajoutant plusieurs propositions (choix 1) et une conclusion (choix 2).
Quand il aura saisi la conclusion, la régle sera affichée à l'écran ainsi que l'ensemble de la base de connaissances.

- Soit Modifier ou supprimer une base de connaissances existante (choix 1)

```
matthieu@matthieu-VirtualBox:~/Bureau/Lo21$ ./main
Bienvenue, que voulez-vous faire ?
 1) Comment ça marche
 2) Créer/modiffier une base de connaissance
 3) Utiliser le moteur d'inférence
 4) Quitter
 2

------------------
Que voulez vous faire ?
 1) Modifier une base deja existante ?
 2) Creer une nouvelle base de connaissance ?
 3) Quitter
1

 Quelles base voulez vous modifier ?
test1
```

Il devra donner le nom de la base à modifier ou supprimer. Le nom qu'il donnera correspondra au nom du fichier qui contient les règles.

Ensuite il pourra modifier (choix2)

```
matthieu@matthieu-VirtualBox:~/Bureau/Lo21$ ./main
Bienvenue, que voulez-vous faire ?
 1) Comment ça marche
 2) Créer/modiffier une base de connaissance
 3) Utiliser le moteur d'inférence
 4) Quitter
 2

------------------
Que voulez vous faire ?
 1) Modifier une base deja existante ?
 2) Creer une nouvelle base de connaissance ?
 3) Quitter
1

 Quelles base voulez vous modifier ?
test1

 Que voulez vous faire ?
 1) Supprimer la base
 2) Ajouter une regle ?
 3) Supprimer une proposition de toutes les regles

2
Ajouter une regle
voici la base avant ajout :

----------------------

Prémisse: a b
Conclusion:  c


Prémisse: b f g
Conclusion:  o


Prémisse: r j p
Conclusion:  m


Prémisse: b o f
Conclusion:  m
```

La base de connaissance avant modification sera affichée.

ou supprimer la base (choix1) si elle existe.

```
matthieu@matthieu-VirtualBox:~/Bureau/Lo21$ ./main
Bienvenue, que voulez-vous faire ?
 1) Comment ça marche
 2) Créer/modiffier une base de connaissance
 3) Utiliser le moteur d'inférence
 4) Quitter
 2

------------------
Que voulez vous faire ?
 1) Modifier une base deja existante ?
 2) Creer une nouvelle base de connaissance ?
 3) Quitter
1

 Quelles base voulez vous modifier ?
test1

 Que voulez vous faire ?
 1) Supprimer la base
 2) Ajouter une regle ?
 3) Supprimer une proposition de toutes les regles

1
la base à bien été supprimer
A bientot
```

Il est également possible de supprimer une proposition de toutes les rêgles de la base de connaissances.

- utiliser le moteur d'inférence (choix 3).

```
matthieu@matthieu-VirtualBox:~/Bureau/Lo21$ ./main
Bienvenue, que voulez-vous faire ?
 1) Comment ça marche
 2) Créer/modiffier une base de connaissance
 3) Utiliser le moteur d'inférence
 4) Quitter
 3


 Veuillez donner le nom de la base de connaissance que vous voullez utiliser (si aucune n'est creer relancer et creer en une):

test


Veuillez maitenant donner vos faits, taper 1 pour ajouter un faits et 2 quand vous avez finit:

5
saisir 1 ou 2


Veuillez maitenant donner vos faits, taper 1 pour ajouter un faits et 2 quand vous avez finit:

a

saisir un nombre


Veuillez maitenant donner vos faits, taper 1 pour ajouter un faits et 2 quand vous avez finit:

1


 Donner un fait:

z

 z a bien été ajouter a vos fait


Veuillez maitenant donner vos faits, taper 1 pour ajouter un faits et 2 quand vous avez finit:

2


 Voici donc votre base de faits

---------------

base de fait :
- z

---------------

Le programme va chercher vos fait :


==> Le moteur d'inférence peut déduire a partire de vos fait et de la base de connaissance les conclusions suivantes :
Aucune conclusion n'a pu être trouvée
```

Il doit alors donner le nom du fichier contenant la base de connaissance qu'il veut utiliser.

Si le nom saisi ne correspond pas à un fichier existant, un message d'erreur sera affiché.

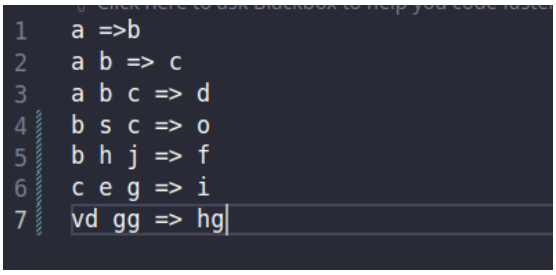
```
matthieu@matthieu-VirtualBox:~/Bureau/Lo21$ ./main
Bienvenue, que voulez-vous faire ?
 1) Comment ça marche
 2) Créer/modiffier une base de connaissance
 3) Utiliser le moteur d'inférence
 4) Quitter
 3


 Veuillez donner le nom de la base de connaissance que vous voullez utiliser (si aucune n'est creer relancer et creer en une):

existepas

=> Erreur lors de l'ouverutre de la base,
Tester avec une autre base
matthieu@matthieu-VirtualBox:~/Bureau/Lo21$
```

Il est obligatoire d'avoir une base de connaissance existante pour pouvoir utiliser le moteur. Une fois la base de connaissance correctement charger, le programme va demander à l'utilisateur de lister les faits. Les faits ne sont pas sauvegardés comme la base de connaissances.

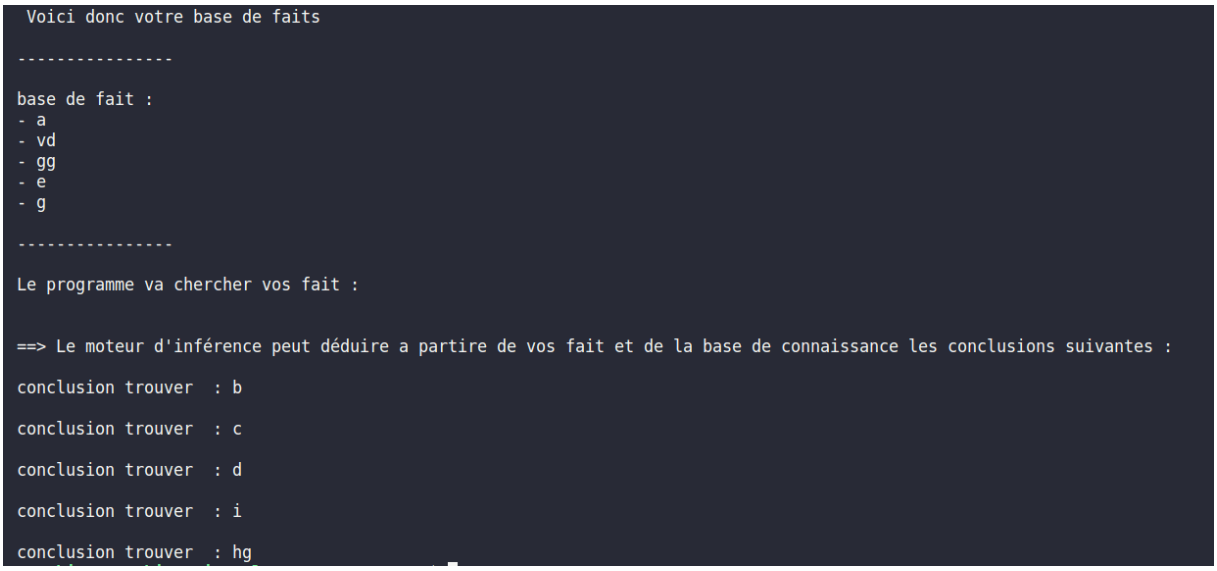
Toute la base de faits est affiché quand il a fini d'insérer les faits.

Finallement, il affichera les conclusions s'il en trouve.

Voici un exemple de fichier de base de connaissances :



voici le résultat qu'on obtient :



Remarque :

il existe également dans notre projet un main2.c qui est plus un test qui montre que les base de connaissance ou les base de faits peuvent être créer directement dans le code du programme

IV. Commentaires sur les résultats

Nous sommes plutôt satisfaits du rendu de notre projet, cependant nous pensons que quelques améliorations devraient être faites. En effet, nous pourrions faire plus de vérifications sur ce que l'utilisateur saisit, par exemple l'empêcher de saisir 2 fois la même proposition dans une règle, 2 fois la même règle dans la base de connaissances ou encore qu'il ne puisse pas donner un règle avec une seule proposition dans la prémisse.
Nous pensons également que nous aurions pu rajouter plus d'options dans le programme, par exemple permettre à l'utilisateur de supprimer une règle de la base déjà créée ou encore rajouter une règle autre par la queue.

Cependant outre ces modifications que nous pourrions apporter, nous somme satisfaits de ce projet. Il nous a permis d'approfondir nos connaissances en C, plus particulièrement sur les listes chainées mais il nous a également beaucoup appris sur l'élaboration d'algorithmes.

Remarque : Nous avons testé le programme uniquement sur Linux.