Projet LO21

Sujet : Système expert de cocktails

Lecas Nicolas

&

Albuzlu Gökdeniz

Semestre : A2020 **Niveau : TC03**

Table des matières

Introduction

Un système expert est composé d'une base de connaissances, une base de faits et un moteur d'inférence.

Dans ce projet nous proposons de réaliser un système expert pour les cocktails dans lequel l'utilisateur est invité à saisir les différents ingrédients de son cocktail au fur et à mesure parmi les ingrédients proposés. En effet, le programme cherchera les ingrédients saisis par l'utilisateurs et lui retournera le nom du cocktail recherché.

Nous allons expliquer dans ce rapport les différents choix de conceptions, les démarches choisies et enfin les difficultés rencontrées. Nous verrons également l'algorithme des sous programmes, leurs explications et enfin l'utilisation du programme.

Remarque : Dans ce projet, il y a des fonctions demandées dans la consigne que nous avons volontairement décidé de ne pas utiliser. Notamment une fonction qui supprime une prémisse d'une règle, notre moteur d'inférence, qui remplit la base de fait et la compare avec les règles de la base de connaissance nous a évité de « supprimer la premisse d'une règle de la base de connaissance jusqu'à ce qu'elle soit vide → la conclusion est vraie ». Nous avons donc choisi une manière de faire différente.
Cependant ces fonctions sont bien codées à la fin du .c , mais jamais appelées.

I) La création d'un type abstrait pour les règles

Tout d'abord nous avons commencé par créer un type abstrait de données appelé **« Liste »** qui nous permettra de créer les règles de la base de connaissance.

```c
/* Définition d'une Liste */
typedef struct elem
{
    char lettres;
    struct elem *next;
}Element;

typedef Element *List
```

En effet cette liste est composée d'une ou plusieurs prémisses et d'une conclusion qui se trouve en fin de liste, c'est-à-dire en queue de liste.

Pour notre cas, on trouve les ingrédients du cocktail avant la fin de la liste et le nom du cocktail en fin de liste. Prenons l'exemple d'un cocktail. En effet le cocktail Monaco est composé de Bière + Sirop de grenadine + Limonade. Donc dans notre exemple, Monaco serait en fin de liste et les ingrédients sont avant.

En effet nous avons choisi d'utiliser 2 fonctions pour remplir nos listes de règles. Une première fonction qui nous sert à insérer la conclusion, c'est-à-dire insérer en fin d'une liste.et une deuxième fonction qui nous permet d'insérer les prémisses, juste avant la queue de liste (en avant dernière position). En effet en utilisant deux fonctions différentes, on réserve la fin de la liste uniquement pour la conclusion.

Enfin nous allons également créer une fonction qui teste si une proposition appartient à la prémisse d'une règle, que nous utiliserons plus tard. Cette fonction prend en paramètre une règle et une proposition. Retournons à notre exemple Monaco cité un peu plus haut, en effet cette fonction va chercher si l'ingrédient saisi par l'utilisateur est présent dans le cocktail Monaco. Si c'est le cas la fonction retourne vrai. D'ailleurs nous avons eu une approche récursive pour cette dite fonction.

Voici l'algorithme de ces sous fonctions

```
Créer la conclusion d une règle

Fonction Inserer_queue(L : Liste, C : chaine de caracteres) : Liste
Début
    Element : newElem <- creerElement()
    lettres(newElem) <- C
    newElem->next <- INDEFINI

    Si est_vide(L) Alors
        Inserer_queue() <- newElem
    Sinon
        Element : Temp <- L
        Tant que non est_vide(Temp->next) Faire
            Temp -> next
        Fait
        Temp->next <- newElem
    Fin Si

    Inserer_queue() <- L
Fin
```

```
Ajouter une proposition à la prémisse d une regle

Fonction Insérer_avant_queue (L : Liste, C : chaine de caracteres) : Liste
Début
    Element : newElem = creerElement()
    lettres(newElem) <- C
    newElem->next <- INDEFINI
    Element : Temp <- L

    Si est_vide(L) Alors
        Insérer_avant_queue() <- newElem
    Sinon Si est_vide(Temp->next) Alors
        L <- Inserer_tete(L,C)
    Sinon
        Tant que non est_vide(Temp->next->next) Faire
            Temp -> next
        Fait

        NewElem->next <- Temp->next
        Temp->next <- newElem
    Fin Si

    Insérer_avant_queue() <- L
Fin
```

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

Fonction est_present(L : Liste, C : chaine de caracteres) : BOOL
Début
    Si est_vide(L) Alors
        est_present() <- Faux
    Sinon
        Si  lettres(L) = C Alors
            est_present() <- Vrai
        Sinon
            est_present() <- est_present(L->next, C)
        Finsi
    Finsi
Fin
```

Une fois que nous avons défini les fonctions. On peut maintenant s'intéresser au remplissage des règles. En effet notre programme comporte 70 cocktails différent ce qui signifie 70 règles différents avec plusieurs prémisses. Ainsi Il était très long et répétitif pour nous de remplir les 70 règles. De ce fait nous avons utilisé un fichier header consacré au remplissage dont nous allons parler plus tard.

Enfin nous avons également utilisé une fonction simple qui permet de tester si une liste est vide ou non, et deux autres fonctions qui retournent la conclusion d'une règle ou la première proposition d'une règle (= d'une liste).

Voici les algorithmes de ces sous fonctions

```
Tester si la prémisse d'une règle est vide

Fonction est_vide(L : Liste) : BOOL
Début
    Si (L = INDEFINI) Alors
        est_vide() <- Vrai
    Sinon
        est_vide() <- Faux
    Finsi
Fin
```

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

Fonction valeurTete(L : Liste) : Liste
Début
    valeurTete() <- L
Fin
```

```
Accéder à la conclusion d'une règle

Fonction Conclusion(L : Liste) : Liste
Début
    Temp=L
    Tant que non est_vide(Temp->next) Faire
        Temp -> next
        Conclusion() <- Temp
    Fait
Fin
```

II) Création d'un type abstrait pour la Base de Connaissances

Après avoir défini un type abstrait Liste pour les règles, nous avons créé un type abstrait Base_connaissance permettant de définir des Bases de Connaissances comme une liste de règles (une liste de listes).

```
/* Définition d'une base de connaissances regroupant toutes les listes */
typedef struct full
{
    List regle_base;
    struct full *next;
}Full;

typedef Full *Base_connaissance
```

En effet c'est ici que nous allons stocker toutes les règles de notre projet. Dans cette partie, la première fonction que nous avons utilisée permet de créer une base de connaissance vide. Ensuite, nous avons créé une fonction qui nous permet de remplir notre base de connaissance, en effet cette fonction prend en paramètre notre base de connaissance et la règle que l'on veut insérer. L'ajout de la règle se fait en queue donc cette fonction est semblable à notre fonction insérer en queue cité plus haut.

Voici l'algorithme qui permet d'insérer une règle dans la base de connaissances

```
Insere un element en fin de Liste dans la base

Fonction Inserer_base (BC : Base de connaissances, regle : Liste) : Base de connaissances
Début
    Full : newElem <- creerElement()
    regle_base(newElem) <- regle
    newElem->next <- INDEFINI

    Si est_vide(BC) Alors
        Inserer_base() <- newElem
    Sinon
        Full : Temp <- BC
        Tant que non est_vide(Temp->next)  Faire
            Temp -> next
        Fait
        Temp->next <- newElem
    Fin Si

    Inserer_base() <- BC
Fin
```

Le remplissage de notre base de connaissance était également très long et répétitif pour nous sachant que nous avons 70 règles à insérer dans la base de connaissance. Nous avons donc utilisé le même fichier Header (« Creation_Base_de_C ») pour le remplissage de notre base ,nous l'avions déjà utilisé pour le remplissage des règles.

D'ailleurs pour nous aider dans cette partie, nous avons également utilisé plusieurs fonctions qui ne sont pas forcément nécessaire pour le fonctionnement de notre programme. Comme la fonction (afficher_base_connaissance) qui nous affiche toutes les règles de notre base de connaissance. En effet cette fonction nous permet d'effectuer des vérifications simples, comme par exemple si notre base se remplit correctement.

```
Fonction qui affiche l ensemble de notre base de connaissances

Fonction afficherBaseConnaissances(BC : Base de connaissances) : Affichage
Début
    Full : Temp <- BC
    Faire
        Element : TempRegleBase <- regle_base(Temp)
        Tant que non est_vide(TempRegleBase) Faire
            Afficher TempRegleBase
            TempRegleBase->next
        Fait
        Temp -> next
    Tant que non est_vide(Temp)
Fin
```

III) Moteur d'inférence

A présent nous avons défini nos règles composées de prémisses et d'une conclusion et notre base de connaissances constitué de nos règles. On utilise donc le type abstrait Liste pour créer notre base de fait. En effet notre base de faits se remplira au fur et à mesure et comportera les ingrédients saisis par l'utilisateur. Une fois le cocktail est trouvé, on l'insère dans notre base de fait.

On peut donc maintenant s'intéresser au moteur d'inférence. Un moteur d'inférence permet de déduire des faits certains (vrais) en partant de la base de faits et en appliquant les règles de la base de connaissance. En effet c'est la partie la plus compliquée et le plus long de notre programme. Pour notre cas, l'utilisateur sera invité à saisir les ingrédients qu'il souhaite dans son cocktail parmi une liste de proposition. En effet nous avons utilisé un système de recherche intelligente. C'est-à-dire que les ingrédients proposés à l'utilisateur sont en fonction des ingrédients qu'il a déjà saisi. Sachant que notre programme comporte plus de 100 ingrédients il était indispensable pour nous d'utiliser cette recherche intelligente.

Reprenons notre exemple avec le cocktail Monaco composé de Bière + Sirop de grenadine + Limonade. En effet si l'utilisateur saisit l'ingrédient bière, alors notre programme lui proposera uniquement les ingrédients des cocktails qui comportent la bière et ne lui proposera pas donc les ingrédients des boissons qui ne comporte pas la bière. Le fait d'utiliser une recherche de ce type nous permet de trier les ingrédients, et simplifie également l'interface de l'utilisateur.

Voici la sous fonction en question, détaillée juste ensuite

```
Fonction qui affiche une reponse intelligente en fonction des choix d ingredients

Fonction intelligence(BC : Base de connaissances, Base_fait : Liste) : Liste
Début
    Liste : listeTemporaire <- nouvelleListe()
    Full : Temp <- BC
    Entier : nombre_de_listes_restantes

    nombre_de_listes_restantes = combien(Base_fait, BC)

    Si nombre_de_listes_restantes != 1 Alors
        Faire
            Element : TempRegleBase <- regle_base(Temp)
            Si correspondance_liste(Base_fait, TempRegleBase) Alors
                Tant que non est_vide(TempRegleBase->next->next) Faire
                    Si non est_present(listeTemporaire, lettres(TempRegleBase->next)) ET non est_present(Base_fait, lettres(TempRegleBase->next)) Alors
                        listeTemporaire = Inserer_queue(listeTemporaire, lettres(TempRegleBase->next))
                    Fin Si
                    TempRegleBase -> next
                Fait
            Fin Si
            Temp -> next
        Tant que non est_vide(Temp)

    Sinon
        Faire
            Element : TempRegleBase <- Temp->regle_base
            Si correspondance_liste(Base_fait, TempRegleBase) Alors
                Tant que non est_vide(TempRegleBase->next) Faire
                    Si non est_present(Base_fait, lettres(TempRegleBase)) Alors
                        Base_fait = Inserer_queue(Base_fait, lettres(TempRegleBase))
                    Fin Si
                    TempRegleBase -> next
                Fait
                listeTemporaire = Inserer_queue(listeTemporaire, lettres(TempRegleBase))
            Fin Si
            Temp -> next
        Tant que non est_vide(Temp)
    Fin Si

    intelligence() <- listeTemporaire
Fin
```

Cette fonction a pour but de renvoyer une liste temporaire qui va servir uniquement d'affichage. Cette liste va comprendre tous les ingrédients à afficher à l'utilisateur en fonction des ingrédients déjà rentrés.

A chaque appel de cette fonction, on vérifie combien de règles dans la base contiennent encore tous les ingrédients entrés par l'utilisateur.

- Si plusieurs règles correspondent, nous n'avons toujours pas trouvé notre cocktail, et nous continuons à afficher les ingrédients restants dans ces dits cocktails afin que l'utilisateur continue sa recherche. La liste temporaire contiendra alors tous ces ingrédients potentiels.

- Si une seule règle correspond, nous avons trouvé notre cocktail et nous procédons à un « remplissage automatique » de la base de faits. En effet prenons un exemple. Dans notre base de connaissances, le « Langue de feu » composé de Bière, Vodka, Tabasco et Piment est le seul cocktail qui contient de la bière et du tabasco. Donc si l'utilisateur entre Bière puis Tabasco, on procède à un remplissage automatique de la base de faits avec Vodka, Piment et la conclusion, Langue de feu. La liste temporaire contiendra alors le nom du cocktail trouvé (la conclusion de la règle = la queue de la liste).

Notre fichier main va récupérer cette liste temporaire, si elle contient un seul élément, on a trouvé le cocktail donc on agit en conséquence, si elle contient plusieurs éléments, le cocktail n'est pas trouvé, ce sont des ingrédients potentiels, on agit en conséquence.

De plus à l'intérieur de cette fonction nous avons utilisé plusieurs sous fonctions. Tout d'abord les fonctions est_vide, est_present et Insérer_queue détaillées plus haut. En plus de celles-ci, on utilise une fonction « combien » qui prend en paramètre notre base de fait et la base de connaissance, cette fonction nous permet de retourner le nombre de règles qui contiennent les ingrédients présents dans la base de fait. Prenons notre exemple de cocktails. Si l'utilisateur saisit l'ingrédient bière, cette fonction nous retournera le nombre de règle comportant la bière dans la base de connaissances. En effet cette fonction est très utile pour notre remplissage automatique.

Une autre fonction que nous avons utilisée est la fonction correspondance _liste. Cette fonction prend en paramètre 2 listes, et vérifie si tous les éléments de la première liste sont présent dans la deuxième liste. Peu importe l'ordre. Revenons à nos cocktails

Si par exemple la première liste comporte Bière + Limonade et que la deuxième liste Comporte Bière + Sirop de grenadine + Limonade. Dans ce cas-là, la fonction nous retourne vrai.

Voici les algorithmes de ces sous fonctions

```
Fonction qui renvoie le nombre de listes de BC qui contiennent tous les elements de Base_fait

Fonction combien(Base_fait : Liste, BC : Base de connaissances) : Entier
Début
    Entier : nombre <- 0
    Full : Temp <- BC
    Faire
        Element : TempRegleBase <- regle_base(Temp)
        Si correspondance_liste(Base_fait, TempRegleBase) Alors
            nombre +1
        Fin Si
        Temp -> next
    Tant que non est_vide(Temp->next)

    combien() <- nombre
Fin
```

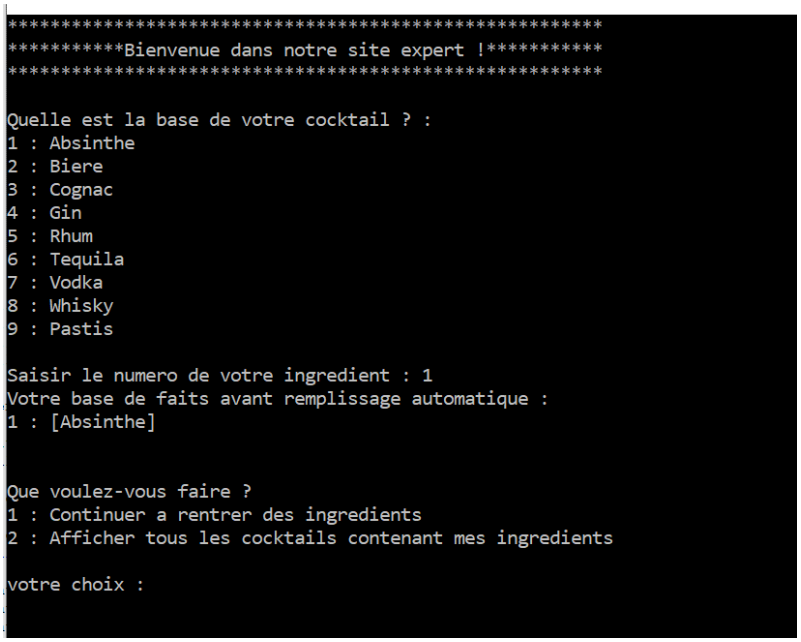
```
Fonction qui renvoie vrai si tous les elements de L1 sont au moins dans L2, faux sinon

Fonction correspondance_liste(L1 : Liste, L2 : Liste) : BOOL
Début
    Element : Temp <- L1
    Si (est_vide(Temp) OU est_vide(L2)) Alors
        correspondance_liste() <- Faux
    Sinon
        Si est_present(L2, lettres(Temp)) Alors
            Tant que est_present(L2, lettres(Temp)) ET non est_vide Temp->next ET est_present(L2, lettres(Temp->next)) Faire
                Temp -> next
            Fait
            Si est_vide(Temp->next) Alors
                correspondance_liste() <- Vrai
            Sinon
                correspondance_liste() <- Faux
            Fin Si
        Sinon
            correspondance_liste() <- Faux
        Fin Si
    Fin Si
Fin
```

IV) Jeux d'essais

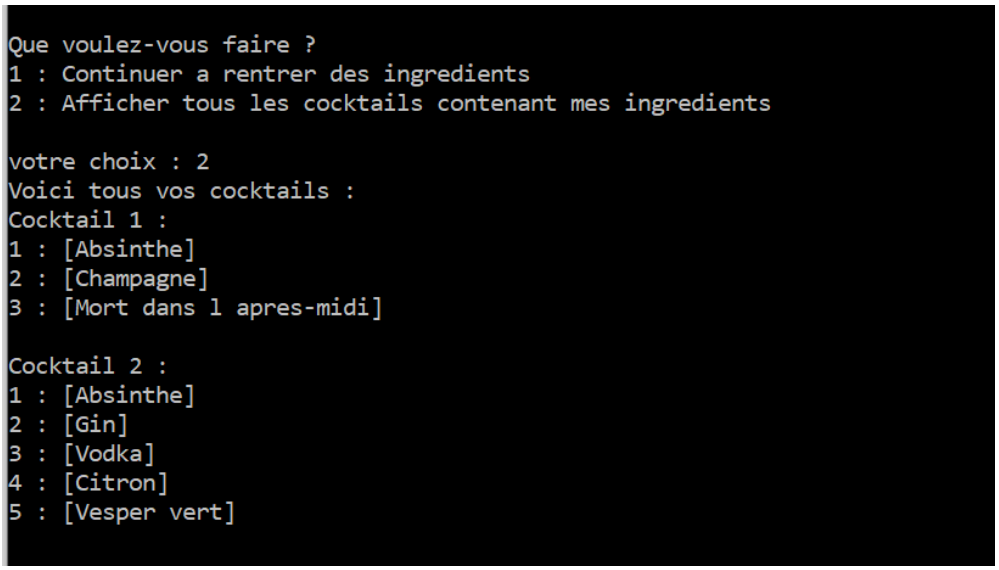
Nous allons maintenant voir le fonctionnement du programme. Tout d'abord lorsque l'utilisateur lance le programme. Il est invité à choisir le premier ingrédient qui compose son cocktail.

Ici Absinthe est notre premier ingrédient, une fois qu'on saisit notre ingrédient voici ce que le programme nous affiche



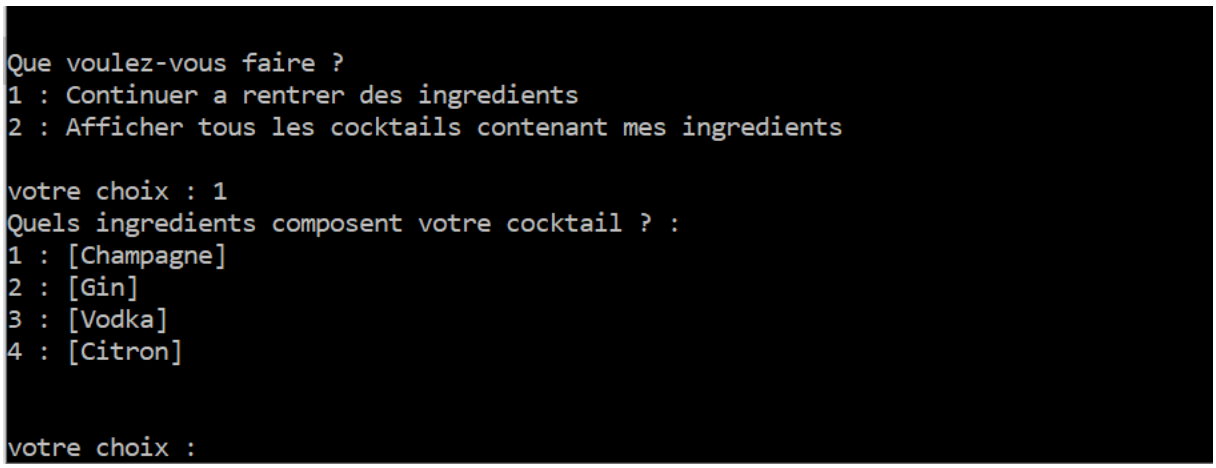
(L'affichage « base de faits avant remplissage automatique » est ici uniquement pour constater le bon fonctionnement de notre fonction de remplissage, cet affichage serait bien entendu indésirable pour l'utilisateur). L'utilisateur possède désormais 2 choix, il peut arrêter le programme en rentrant le « 2 » en effet cette fonction affichera tous les cocktails contenant les ingrédients déjà saisis par l'utilisateur dans la base de fait.

Essayons maintenant ces fonctions dans notre cas



Ici l'utilisateur à décider d'afficher tous les cocktails contenant les ingrédients déjà présents dans sa base de fait (choix 2). En effet le programme lui affiche les cocktails qui comportent l'absinthe, le dernier élément des listes correspondent toujours aux noms des cocktails.

Revenons en arrière et supposons que l'utilisateur continue à saisir (choix 1)



Ici, on peut remarquer que les ingrédients qui sont proposé à l'utilisateur sont en lien avec l'ingrédient Absinthe qu'il a déjà saisi en premier.

Supposons que l'utilisateur choisisse maintenant l'ingrédient « Citron » (choix 4) comme deuxième ingrédient.

```
votre choix : 4

Votre base de faits avant remplissage automatique :
1 : [Absinthe]
2 : [Citron]


Votre base de faits apres remplissage automatique :
1 : [Absinthe]
2 : [Citron]
3 : [Gin]
4 : [Vodka]
5 : [Vesper vert]


Le cocktail que vous recherchez est le : Vesper vert
-----Vous avez trouve votre cocktail !-----

Maintenant que voulez vous faire ?
1 : Quitter
2 : Chercher un autre cocktail

votre choix :
```

Ici dans notre cas le programme trouve directement le cocktail correspondant, en lui affichant le nom du cocktail (ici « Vesper Vert »). En effet on à trouvé directement le cocktail avant même de demander les autres ingrédients qui composent le cocktail, car les ingrédients saisis correspondent avec une seule règle de la base de connaissance. Cette approche est beaucoup plus efficace pour l'utilisateur et elle simplifie également l'interface et l'utilisation du programme.

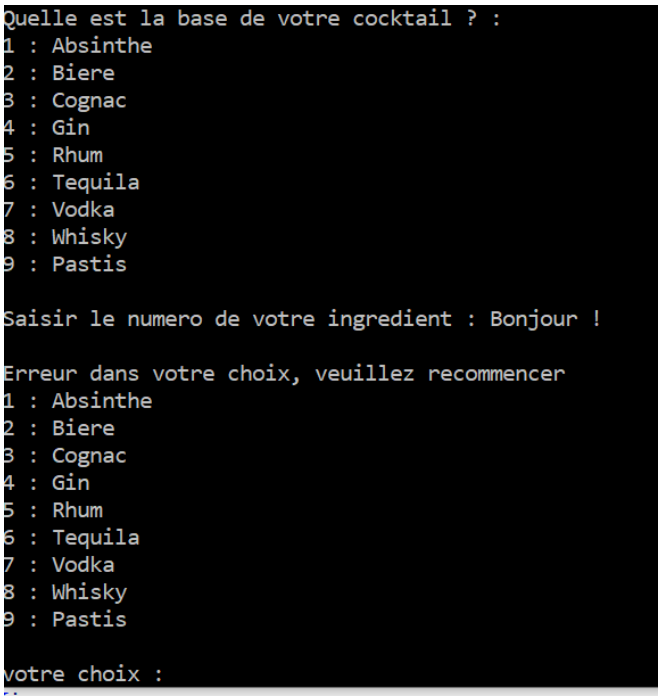
Une fois que le programme est fini l'utilisateur peut recommencer le programme ou bien il peut quitter. Dans les deux cas on fait appel à nos fonctions pour nettoyer les listes que nous avons utilisé, en effet les listes temporaires, la base de connaissance et notre base de fait sont vidées.

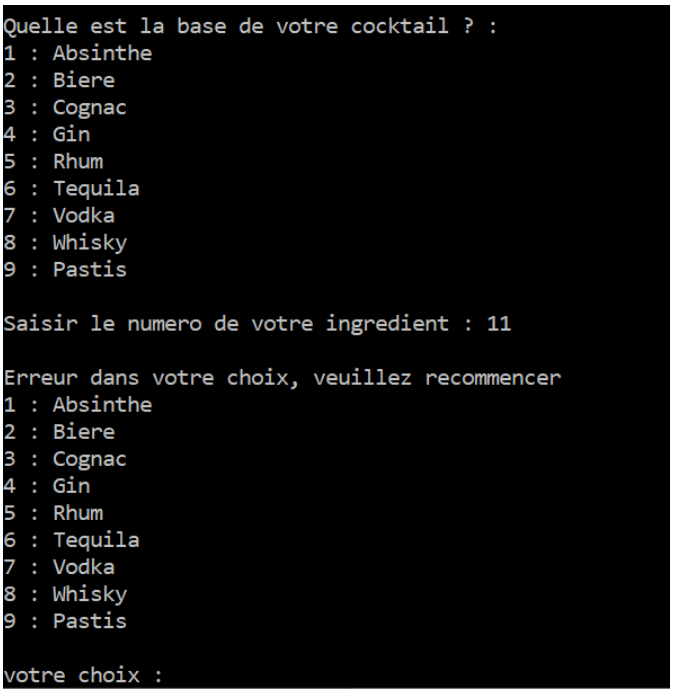
```
Fonction de nettoyage de tout ce qu on a utilisé en fin de programme

Fonction nettoyage(BC : Base de connaissances, BF : Liste, LT : Liste) : void
Début
    Full : Temp <- creerFull()
    Element : Temp1 <- creerElement()
    Element : Temp2 <- creerElement()

    Tant que non est_vide(BC) Faire
        Temp <- BC->next
        liberer(BC)
        BC<- Temp
    Fait
    Tant que non est_vide(BF) Faire
        Temp1 <- BF->next
        liberer(BF)
        BF<- Temp1
    Fait
    Tant que non est_vide(LT) Faire
        Temp2 <- LT->next
        liberer(LT)
        LT<- Temp2
    Fait
Fin
```

Finalement, nous avons également sécurisé au maximum notre programme. En effet l'utilisateur peut saisir absolument n'importe quels caractères, notre programme va lui indiquer une erreur et lui sommer de recommencer. Voici quelques exemples de sécurité.





L'utilisateur à saisi « Bonjour ! » et le programme affiche un message d'erreur.

De la même manière si l'utilisateur à saisie un faux numéro d'ingrédient ou quoi que ce soit d'autre, un message d'erreur est affiché, et ceci pour chaque étape du programme. Les procédés de sécurité sont détaillés dans le code (main.c : ligne51)

CONCLUSION

Nous avons rencontré plusieurs difficultés durant ce projet, cependant, nous sommes heureux de l'avoir terminé entièrement, en effet ce projet nous a permis de mieux comprendre l'importance de l'utilisation des listes chainées et de la récursivité dans les programmes. De plus, le choix de ce sujet nous a permis d'apprendre plusieurs cocktails différents. Nous remercions l'ensemble des professeurs de LO21 pour nous avoir épaulé tout au long de ce projet.

Sources

Provenance de nos règles (nom des cocktails + ingrédients) :

https://fr.wikipedia.org/wiki/Liste_des_cocktails_par_type_d%27alcool