# Rapport de Projet de Compilation

Fastfood

Heqing HUANG
Yifan ZHANG

14 decembre 2015

# TABLE DES MATIÈRES

# INTRODUCTION

## La fonction du programme

### *Contexte*

Un vendeur de restauration rapide propose 5 sandwichs : **Fromage, Jambon-beurre, Panini, Belge, Dieppois,** dont les compositions sont parmi des 12 types de ingrédients.

1. Les clients peuvent y rajouter des ingrédients parmi la liste suivante : *pain, jambon (1 tranche), beurre (10g), salade (10g), emmental (2 tranches), ketchup (10g), moutarde (10g), mayonnaise (20g), frites (50g), tomate (1), steak, thon (50g)*.

2. Les clients peuvent demander la suppression d'un ingrédient présent. Ils peuvent aussi demander le doublement de la portion d'un ingrédient déjà présent. Tout sandwich contient *au maximum une viande (jambon, steak, thon)*. L'ajout, le doublement ou la suppression d'un ingrédient est facturé **0,50€**.

### *La fonction principale*

Ce programme prend en entrée un liste de commande de sandwichs, comme par exemple

```
1 belge
3 paninis dont 2 avec frites mais sans jambon et 1 sans tomate
3 belges dont 1 sans steak mais avec double frites
```

et produit ensuite plusieurs comptes-rendus :

— une facture détaillée ;

— une liste agrégée des sandwichs à préparer pour la cuisine ;

— une liste d'ingrédients utilisés pour mettre à jour l'inventaire.

## La structure du programme

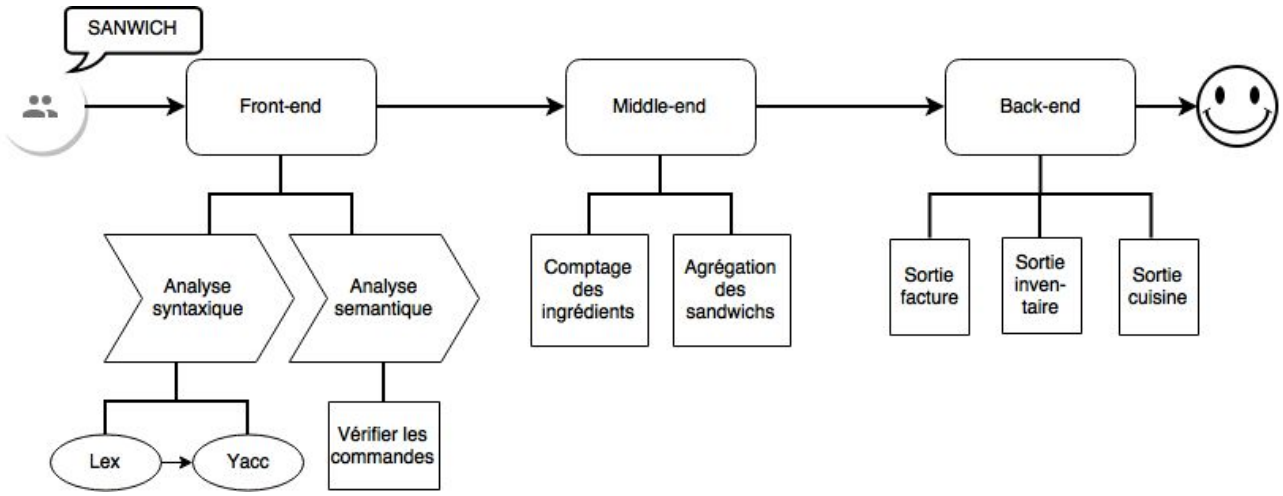



***fig.1. La structure principale***

Lorsque les clients fait une commande, on va analyser ces commandes, calculer les fatures, calculer les inventaires et faire une liste pour la cuisine. On divisier ce programme à 3 parties suivantes:

### *Front-ends*

Cette partie est de recoonaître un fichier qui contient des commandes en langue naturelle. Certaines fichiers doivent être refusés, soit parce que leur syntaxe est incorrecte, soit parce qu'ils ne respectent pas les règles pour la confection des sandwichs (exemple : demande de suppression d'un ingrédient non présent). On doit faire la vérification pour les commaneds dans cette partie.

**Analyse syntaxique :** Cette première partie consiste à analyser le fichier pour en produire **un arbre de syntaxe abstraite** *(liste de commandes)*. Dans cette partie, on utilise *les et bison.*

**Analyse sémantique :** Il s'agit ici de **vérifier** que les sandwichs commandés respectent bien les règles du restaurant :

— Les clients ne peuvent enlever qu'un ingrédient déjà présent.

— Les clients ne peuvent pas ajouter un ingrédient déjà présent.

— Un ingrédient ne peut être doublé que s'il est déjà présent.

— Les clients ne peuvent avoir qu'une seule viande (jambon, steak, thon).

### *Middle-ends*

Cette partie traite le AST et sortie en quelques certains formats.

**Comptage des ingrédients :** Pour la sortie vers l'inventaire, il faut compter, pour chaque ingrédient, quelle quantité est nécessaire pour l'élaboration de la commande.

**Aggrégation des sandwich :** Pour la sortie vers la cuisine, il faut regrouper les sandwichs du même nom ensemble, en les détaillant par version. La **version** d'un sandwich est donnée par l'ensemble (au sens mathématique) des modifications qui lui sont apportées. Ainsi, un panini avec moutarde mais sans jambon sera de la même version qu'un panini sans jambon mais avec moutarde. (L'ordre des modifications n'a pas d'importance.) Par contre un panini sans emmental ni tomate sera différent d'un jambon-beurre sans beurre.

### *Back-ends*

Le but de cette partie est de afficher le texte en format certain, (par exemple, facture et inventaire) et créer un fichier *.html* pour affichage.

**Facture :** Le but de cette partie est de produire un fichier texte qui correspond à la facture de la commande. Ce fichier sera constitué d'un certain nombre de blocs, puis une ligne pour le total. Chaque bloc sera constitué d'une ligne contenant le nombre de sandwichs et son nom avec leur prix de départ, puis d'autant de lignes qu'il y a de modifications, avec leur tarif.

**Inventaire :** Le but de cette partie est de sortir un tableau au format CSV constitué d'une colonne contenant le nom des ingrédients en face desquels on trouve la quantité nécessaires à la réalisation de la commande.

**Cuisine :** Le but de cette partie est de produire une page *HTML* qui recense l'ensemble des sandwichs à réaliser, regroupés par sorte. Le corps de cette page contiendra une suite de blocs constitués :

— D'un titre de niveau 1 *(<h1>)* contenant le nom du sandwich

— D'une liste non énumérée *(<ul>)* dont chaque item correspondra à une version, et contiendra le nombre de sandwichs de cette version et une description en langage naturel de la version.

# DETAILS TECHNIQUES

On a choisit de faire le projet en C. On a divisé ce projet à trois parties: *Front-ends, Middle-ends et Back-ends*. *Front-ends* concerne de l'analyse syntaxe (*flex et bison*) et l'analyse sémantique. *Middle-ends* s'agit de traiter les données en certaines formes, comme Comptage des ingrédients et Agrégation des sandwichs. Dans le *Back-ends*, les données sont affiché sur la sortie standard en plusieurs formes différentes.

On va présenter les trois parties plus précisément dans le rapport suivant.

## Front-ends

### Structures principales

Les commandes sont analysées dans cette partie et le programme transforme ces commandes aux arbres syntaxes abstraites respectivement.

Pour créer l'arbre syntaxe abstraite, on distingue 5 type de noeud dans l'arbre syntaxe abstraite:

—**split** : les mots qui sert à séparer les types différents de sandwich, comme *"dont", "et"*;

—**number** : les nombres des ingrédients et des sandwichs;

—**ingredient** : les noms des ingrédients;

—**operation** : les mots qui sert à ajouter certaines conditions des sandwichs, comme *"avec", "sans", "ni", "et", ","*;

—**entity** : on stocke rien dans ce type de noeud, il représente un type de sandwich.

On a défini certaines structures pour créer ce programme, et ces structures sont les suivantes:

La structure `union` ast est le type de contenu de noeud de l'arbre syntaxe abstraite, il soit type de `int` quand on stocke les nombres des ingrédients ou des sandwichs, soit type de `char*` quand on stocke les opérations ou les noms d'ingrédient ou les noms de sandwich.

```
union ast {
     int num;
     char* word;
```

```
};
```

La structure `node` est le type de noeud de l'arbre syntaxe abstraite, elle contient le type de ce noeud, le contenu et les sous-arbres. On utilise cette structure principalement pour créer l'arbre syntaxe abstraite.

```
typedef struct node {
     int typenode;
     union ast content;
     struct node* left;
     struct node* right;
}node;
```

La structure `commandes` est le pointeur qui pointe sur son arbre syntaxe abstraite, elle contient le nom de ce sandwich et le pointeur de l'arbre syntaxe abstraite.

```
typedef struct commandes {
     char* type;
     node head;
}commandes;
```

La structure `kind` est un type de sandwich, elle contient les conditions spéciales d'un types de sandwich(`require`), le nombre de ce type de sandwich(`cnt`) et le nombre des conditions(`num`).

```
typedef struct kind {
     char** require;
     int cnt;
     int num;
}kind;
```

La structure `version` représente une version d'un sandwich, elle contient le nom de ce sandwich, ses types différents(`types`) et le nombre des types.

```
typedef struct version {
     char* type;
     kind* types;
     int num;
}version;
```

## Analyse syntaxe

Dans cette partie, on utilise flex et bison pour analyser une commande et la transformer à un arbre syntaxe abstraite correspondant.

### *Flex : analyse lexique*

Tout à bord, on crée un analyseur lexcal à fin de reconnaître des mots et émet des tokens à l'analyseur syntaxique. Dans le fichier "fastfood.l", il faut associer tous les mots du Pseudo-Pascal à un token. Il y a 5 types de token de retour dans cette partie.

Dans le fichier "fastfood.l", on définit quelques patterns pour reconnaître et classifier les mots différents, transmettre la valeur qui associe au pattern à bison et retourner un token qui associe à bison.

### *Bison : analyse syntaxe*

Dans cette partie, on analyse les mots qui sont déjà été analysé par analyseur lexque et on les transmets à un arbre syntaxe abstraite par l'automate suivant :

```
Programme ——→ Programme Condition '\n'
Condition ——→ Simple SPLIT Expr | Simple Taste | Simple
     Expr ——→ NUMBER Taste SPLIT Expr | NUMBER Taste
    Taste ——→ OPERATION INGREDIENT | OPERATION INGREDIENT Taste
   Simple ——→ NUMBER TYPE
   NUMBER ——→ [1-9]+[0-9]*
    SPLIT ——→ et | dont
OPERATION ——→ avec | sans | ni | et | ,
     TYPE ——→ fromage | jambon-beurre | panini | belge | dieppois
INGREDIENT ——→ pain | jambon | beurre | salade | emmental | ketchup | moutarde
               | mayonnaise | frites | tomate | steak | thon
```

***fig.2. L'automate du programme***

Les règles de création de l'arbre syntaxe abstriate sont les suivants:

1. Tous les nombre doivent être dans le sous-arbre à gauche.
2. Lorsqu'il y a des opérations ou des splits, on les mets dans le sous-arbre à droite et les entité ou les ingrédients sont dans le sous-arbre à gauche correspondante.

3. Jusqu'à le programme lit le dernière opération ou le dernier split, le dernier ingrédient sera mis à gauche toujours, mais la entité sera mis à droite.
4. S'il y a "mais", on prend "mais" avec l'opération suivante pour un ensemble, comme l'opérateur qui associent à droite.
5. S'il y a "double", on prend "double" avec l'opération précédent pour un ensemble, comme l'opérateur qui associent à gauche.

Pour une mieux intérpretation des règles de création de l'arbre syntaxe abstriate, je donne plusieurs exemples:

*exemple le plus simple,*

```
2 paninis
```

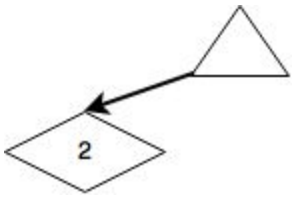



***fig.3.1. L'arbre syntaxe abstraite(exemple le plus simple)***

*exemple en niveau 1,*

```
1 panini avec ketchup, moutarde et mayonnaise
```

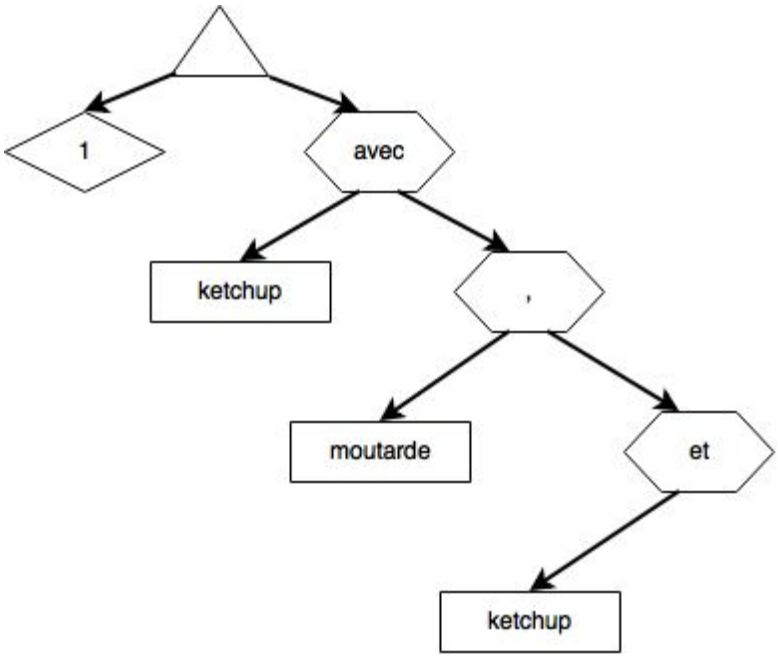



***fig.3.2. L'arbre syntaxe abstraite(exemple en niveau 1)***

*exemple en niveau 2,*

`1 belge sans steak, frites ni salade mais avec tomate et ketchup`

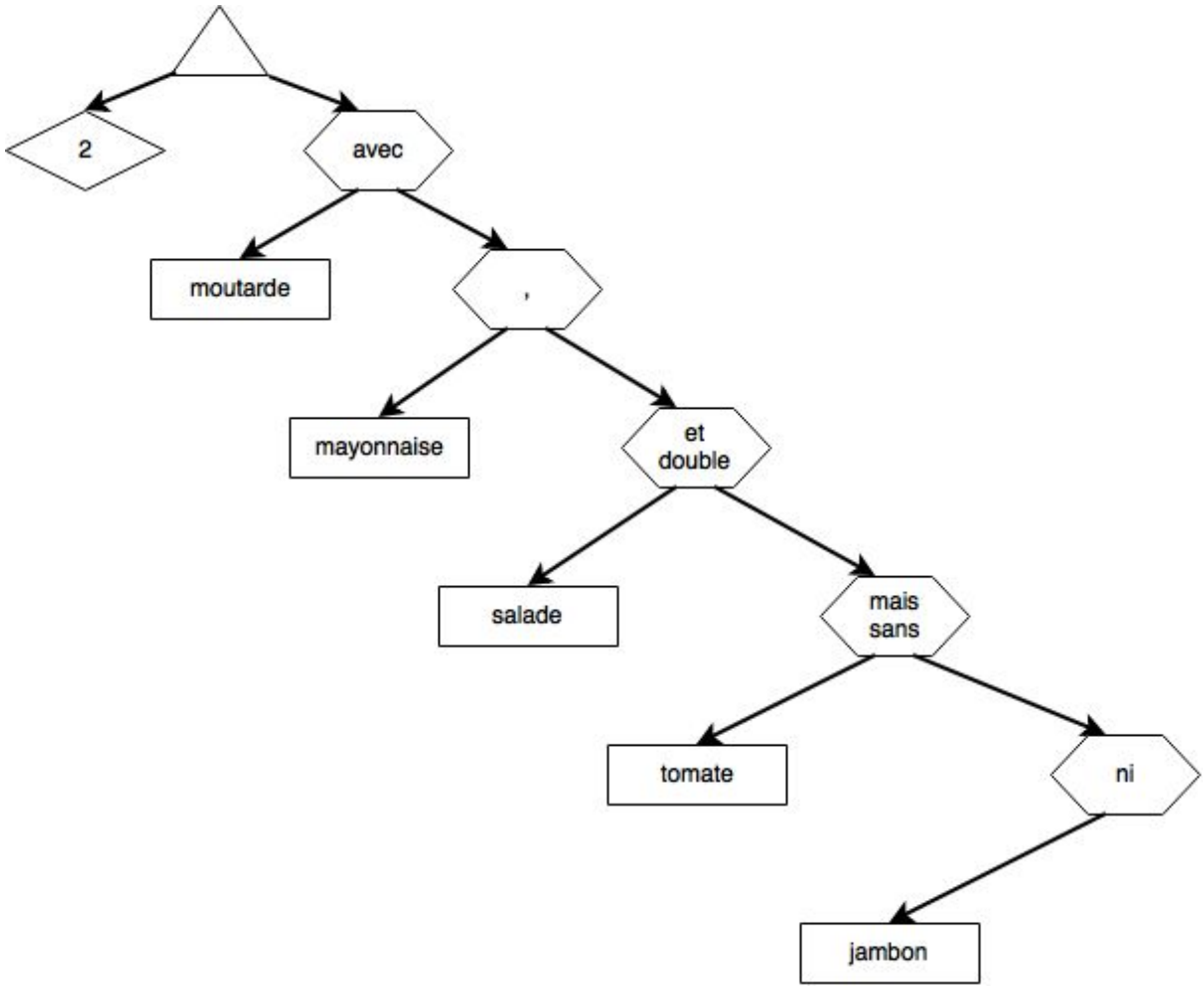



***fig.3.3. L'arbre syntaxe abstraite(exemple en niveau 2)***

*exemple plus complexe (en niveau 3),*

`5 dieppois dont 1 avec moutarde et ketchup et 1 avec double salade, frites et emmental et 2 avec beurre, moutarde et steak mais sans thon ni salade`

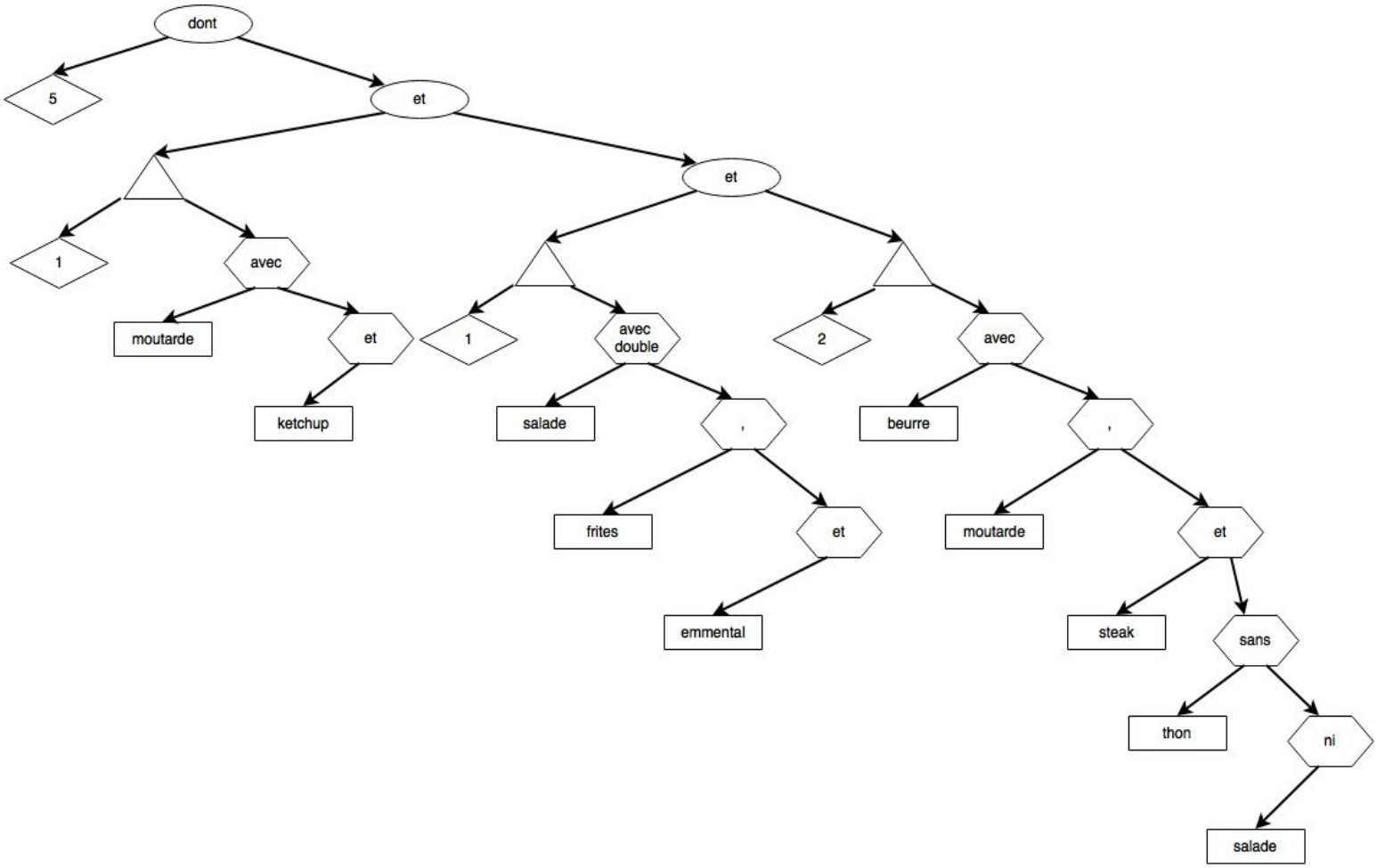



*fig.3.4. L'arbre syntaxe abstraite(exemple en niveau 3, plus complexe)*

## Analyse sémantique

Cette partie sert à vérifier que les sandwichs commandés respectent bien les règles du restaurant.

La fonction `verifie_commande` prend en argument d'une liste de commandes (arbre syntaxe abstraite) et retourne vrai si la commande est valide par rapport aux règles.

On analyse chaque commande, et vérifie si cette commande bien respecte les règles. Si oui, on affiche rien et retourne vrai, sinon, on affiche message d'erreur sur la sortie standard et retourne false.

On cite une liste des ingrédients d'un type. S'il y a *double* ou *sans*, on les compare avec la liste de ce type de sandwich pour vérifier s'li existe dans la liste des ingrédients de ce type de sandwich. S'il y a *avec*, on les compare aussi avec la liste de ce type de sandwich

pour vérifier s'il n'existe pas dans la liste. Pour la quantité de viande, on construit une variable globale pour vérifier le quantité de viande.

## Middle-ends

À cause de la structure de l'arbre syntaxe abstraite, on utilise parcours préfixe pour transformer une commande à une type de version, et les stocke dans une tableau de version. Dans les parties suivantes, on utilise plutôt la structure de version pour manipuler.

### Transformation

Pour transformation de commande, on utilise 4 fonctions :

```
char** collect_require(node* point, char** res, char* opr);
```

On extrait touts les conditions d'un type de sandwich, et on les regroupe à un tableau de longueur deux, on stocke les conditions qui commence par "avec" pour le premier élément dans ce tableau, et les conditions qui commence par "sans" pour le deuxième élément dans ce tableau.

```
kind* make_kind(node* head);
```

on construit une variable qui est le type de kind, et contient les conditions qui sont déjà regroupé dans la fonction `collect_require`.

```
kind* collect_kind(node* point, kind* res);
```

On écrit cette fonction pour regrouper touts les types de kind qui est construit dans la fonction `make_kind`, et les stocke dans un tableau.

```
version transform(node* head, char* type);
```

on construit une variable de type version, qui contient le tableau qu'on peut obtenir par la fonction `collect_kind`.

### Comptage des ingrédients

Dans cette partie, pour calculer les nombres des ingrédients, on définit quelques structures :

La structure ingredient contient le nom de l'ingrédient et son nombre.

```
typedef struct ingredient {
     float num;
     char* name;
}ingredient;
```

La structure cook représente comme recettes d'un sandwich, elle contient le nom du sandwich et une liste des ingrédients de ce type de sandwich.

```
typedef struct cook {
     char* name;
     ingredient material[5];
}cook;
```

D'abord, on stocke le recette de chaque type de sandwich dans un tableau de cook, et définit une tableau de tous les ingrédients dont tous les quantités sont initialisé à 0 au début. En même temps, on stocke les quantités standards de chaque ingrédient dans un tableau de ingrédient.

Ensuite, on compare le type de sandwich dans la commande courante avec le tableau de cook. On peut trouver sa recette et ajouter les quantités de chaque ingrédient dans le tableau de ingrédient.

Finalement, on prend garder à les conditions spéciales de ce sandwich, par exemple, avec tomate, sans jambon etc. Il faut les considérer, donc, on analyse ces conditions et ajoute ou déduire certains quantités des ingrédients dans le tableau de ingrédient.

### Agrégation des sandwichs

Pour la sortie vers la cuisine, il faut regrouper les sandwichs du même nom ensemble, en les détaillant par version. La version d'un sandwich est donnée par l'ensemble (au sens mathématique) des modifications qui lui sont apportées. Ainsi, un panini avec moutarde mais sans jambon sera de la même version qu'un panini sans jambon mais avec moutarde. (L'ordre des modifications n'a pas d'importance.) Par contre un panini sans emmental ni tomate sera différent d'un jambon-beurre sans beurre.

On écrit une fonction qui prends d'une liste de commandes en entrée et qui produit une liste contenant des couples constitués d'un nom de sandwich et d'une liste de couples constitués d'une version et d'un entier indiquant le nombre de tels sandwichs commandés.

On utilise deux fonctions pour réaliser cette partie: `combine_types` et `combination`.

La première fonction `combine_types` concerne de combinaison des versions des sandwichs dont les noms sont pareils mais leurs versions sont différentes. Elle retourne un nouveau tableau de version qui est été combinée.

La deuxième fonction `combination` s'agit de comparer les différents types de version, s'ils sont pareils, on les combiner, sinon, on les laisse. Dans cette fonction, on appelle la fonction précédente. On utilise le résultat de la fonction précédente et on combine les types pareils dans une même version. On retourne le nouveau tableau de version.

## Back-ends

### Facture

Dans cette partie, on calcule les tarifs. Pour ce but, on définit une nouvelle structure: Cette structure contient le nom du sandwich et son prix.

```
struct price {
     char* name;
     float euro;
};
```

On stocke tous les prix de sandwich dans un tableau de price pour consulter leur prix plus facilement.

Après comparaison le sandwich dans le tableau et le sandwich courant, on peut obtenir le prix de ce type de sandwich. Puis, on analyse les conditions spéciales et les compte, le client doit payer (0.5 * nombre de condition) supplémentaire. Donc le tarif total pour ce commande est `cnt` qui est stocké dans la commande fois la somme du prix et le tarif supplémentaire. Finalement, on peut calculer le tarif total par sommation des tarif de chaque commande.

On utilise plusieurs boucles pour réaliser cette forme de sortie.

## Inventaire

Dans cette partie, on calcule les quantités de chaque ingrédient pour réaliser cette commande, et on les affiche. On peut utiliser le résultat de Middle-ends directement.

Le but de cette partie est de sortir un tableau au format CSV constitué d'une colonne contenant le nom des ingrédients en face desquels on trouve la quantité nécessaires à la réalisation de la commande.

## Cuisine

Dans cette partie, on utilise les fonctions (comme `open()` et `write()`) pour créer et modifier le fichier `cuisine.html` qui s'agit de affichage dans une web page.

On utilise les résultats de Middle-ends, et les commandes sont déjà été combinées aux plusieurs types des versions.

# TEST

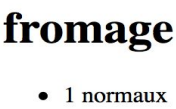
On développe ce programme sous **Mac OS**, donc touts les tests sont faits sous Mac OS. On teste touts les exemples dans sujet, et notre programme fonctionne bien sous **Mac OS**. Voici quelques résultats des exemples aux niveaux différents.

## Test sous Mac OS

exemple en niveau 1(le plus simple: `1 fromage`):

```
waiting for the new command
1 fromage
id: 1  sandwich: fromage

printing facture
 1 fromage                    3.00
                   Total :   3.00
begin inventaire
pain,1.00
jambon,0.00
beurre,10.00
salade,0.00
emmental,2.00
ketchup,0.00
moutarde,0.00
mayonnaise,0.00
frites,0.00
tomate,0.00
steak,0.00
thon,0.00
waiting for the new command
```

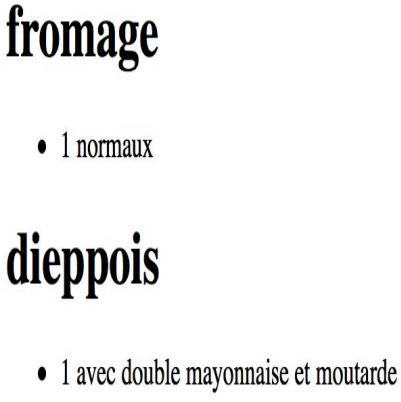



(page web site)

On peut taper une nouvelle commande

(`1 fromage, 1 dieppois avec double mayonnaise et moutarde`):

```
waiting for the new command
1 dieppois avec double mayonnaise et moutarde
id: 1  sandwich: dieppois
 operator: avec double  ingredient: mayonnaise  operator: et  ingredient: moutarde
printing facture
 1 fromage                3.00
 1 dieppois               4.50
   1 avec double mayonnaise et moutarde 1.00
                Total :   8.50
begin inventaire
pain,2.00
jambon,0.00
beurre,10.00
salade,10.00
emmental,2.00
ketchup,0.00
moutarde,10.00
mayonnaise,40.00
frites,0.00
tomate,0.00
steak,0.00
thon,50.00
waiting for the new command
```



exemple en niveaux 2

(la précédente plus `1 belge avec tomate mais sans mayonnaise`):

```
waiting for the new command
1 belge avec tomate mais sans mayonnaise
id: 1  sandwich: belge
 operator: avec  ingredient: tomate  operator:
printing facture
 1 fromage                    3.00
 1 dieppois                   4.50
   1 avec double mayonnaise et moutarde 1.00
 1 belge                      5.00
   1 avec tomate               0.50
   1 sans mayonnaise           0.50
                  Total :    14.50
begin inventaire
pain,3.00
jambon,0.00
beurre,10.00
salade,20.00
emmental,2.00
ketchup,0.00
moutarde,10.00
mayonnaise,40.00
frites,50.00
tomate,0.50
steak,1.00
thon,50.00
waiting for the new command
```

# fromage

- 1 normaux

# dieppois

- 1 avec double mayonnaise et moutarde

# belge

- 1 avec tomate mais sans mayonnaise

(`1 belge avec thon mais sans steak`)

```
waiting for the new command
1 belge avec thon mais sans steak
id: 1  sandwich: belge
 operator: avec  ingredient: thon  operator: ma
printing facture
 1 fromage                    3.00
 1 dieppois                   4.50
   1 avec double mayonnaise et moutarde 1.00
 1 belge                      5.00
   1 avec tomate               0.50
   1 sans mayonnaise           0.50
 1 belge                      5.00
   1 avec thon                 0.50
   1 sans steak                0.50
                  Total :    20.50
begin inventaire
pain,4.00
jambon,0.00
beurre,10.00
salade,30.00
emmental,2.00
ketchup,0.00
moutarde,10.00
mayonnaise,60.00
frites,100.00
tomate,0.50
steak,1.00
thon,100.00
```

# fromage

- 1 normaux

# dieppois

- 1 avec double mayonnaise et moutarde

# belge

- 1 avec tomate mais sans mayonnaise
- 1 avec thon mais sans steak

exemple en niveau 3

(la précédente plus `2 paninis dont 1 sans jambon et 1 avec ketchup`)

```
waiting for the new command
2 paninis dont 1 sans jambon et 1 avec ketchup
id: 2  sandwich: paninis
 splite: dont  id: 1  operator: sans  ingredie
printing facture
 1 fromage                    3.00
 1 dieppois                   4.50
   1 avec double mayonnaise et moutarde 1.00
 1 belge                      5.00
   1 avec tomate               0.50
   1 sans mayonnaise           0.50
 1 belge                      5.00
   1 avec thon                 0.50
   1 sans steak                0.50
 2 paninis                   10.00
   1 sans jambon               0.50
   1 avec ketchup              0.50
                  Total :    31.50
begin inventaire
pain,6.00
jambon,1.00
beurre,10.00
salade,50.00
emmental,6.00
ketchup,10.00
moutarde,10.00
mayonnaise,60.00
frites,100.00
tomate,1.50
steak,1.00
thon,100.00
```

# fromage

- 1 normaux

# dieppois

- 1 avec double mayonnaise et moutarde

# belge

- 1 avec tomate mais sans mayonnaise
- 1 avec thon mais sans steak paninis

# paninis

- 1 sans jambon
- 1 avec ketchup

(`3 paninis dont 1 sans jambon ni tomate mais avec ketchup et 1 avec double salade`)

```
waiting for the new command
3 paninis dont 1 sans jambon ni tomate mais av
id: 3  sandwich: paninis
 splite: dont  id: 1  operator: sans  ingredie
dient: salade
printing facture
 1 fromage                    3.00
 1 dieppois                   4.50
   1 avec double mayonnaise et moutarde 1.00
 1 belge                      5.00
   1 avec tomate               0.50
   1 sans mayonnaise           0.50
 1 belge                      5.00
   1 avec thon                 0.50
   1 sans steak                0.50
 2 paninis                   10.00
   1 sans jambon               0.50
   1 avec ketchup              0.50
 3 paninis                   15.00
   1 avec ketchup              0.50
   1 sans jambon ni tomate     1.00
   1 avec double salade        0.50
                  Total :    48.50
begin inventaire
pain,9.00
jambon,3.00
beurre,10.00
salade,90.00
emmental,12.00
ketchup,20.00
moutarde,10.00
mayonnaise,60.00
frites,100.00
tomate,2.50
steak,1.00
thon,100.00
```

# fromage

- 1 normaux

# dieppois

- 1 avec double mayonnaise et moutarde

# belge

- 1 avec tomate mais sans mayonnaise
- 1 avec thon mais sans steak paninis

# paninis

- 1 sans jambon
- 1 avec ketchup
- 1 mais avec ketchup sans jambon ni tomate
- 1 avec double salade
- 1 normaux

## Test sous Linux

On les teste aussi sous système Linux Ubuntu, mais pour certaines exemples, il y a quelques problèmes sur allocation du mémoire en utilisant la fonction `realloc()`.

Un des ces exemples est `1 belge sans steak, frites ni salade mais avec tomate et ketchup`.

On tape cette commande (`1 belge sans steak, frites ni salade mais avec tomate et ketchup`) dans Linux:

```
waiting for the new command
1 belge sans steak, frites ni salade mais avec tomate et ketchup
id: 1  sandwich: belge
 operator: sans  ingredient: steak operator: ,  ingredient: frites  operator: ni
  ingredient: salade  operator: mais avec  ingredient: tomate  operator: et  ing
redient: ketchup
*** Error in `./fastfood': realloc(): invalid next size: 0x000000000098c680 ***
Aborted (core dumped)
```

Il y a erreur sur realloc() lorque l'allocation du memoire, mais la même commande sous Mac OS, ça fonctionne bien:

```
waiting for the new command
1 belge sans steak, frites ni salade mais
id: 1  sandwich: belge
 operator: sans  ingredient: steak operato
printing facture
 1 belge                    5.00
   1 avec tomate et ketchup   1.00
   1 sans steak , frites ni salade 1.50
                 Total :   7.50
begin inventaire
pain,1.00
jambon,0.00
beurre,0.00
salade,0.00
emmental,0.00
ketchup,10.00
moutarde,0.00
mayonnaise,20.00
frites,0.00
tomate,0.50
steak,0.00
thon,0.00
waiting for the new command
```

On ne trouve pas la raison de ce problème, mais pour les autres exemples sauf ces certaines exemples, ça fonctionne bien dans Linux.

# APPORTS ET LIMITES

## Apports

La réalisation de ce programme a été une véritable mise en situation pratique des compétences techniques acquises lors de la formation de cour :

1. On choisit `lex` et `bison` pour réaliser ce programme, donc on connaît mieux l'utilisation de `lex` et `bison`.
2. On utilise les connaissances du cours, et on les met en pratique, ça nous aide de comprendre mieux des connaissances du cours.
3. On comprend mieux l'utilisation de l'automate et arbre syntaxe abstraite.
4. On a approfondi notre compétence en compilation.

## Limites

On développe ce programme sous **Mac OS**, le limite de ce programme est l'utilisation sous **Linux**. Il y a des erreurs sur certaines exemples. Lorsque le programme appelle la fonction `realloc()`, il y a des erreurs d'allocation du mémoire. Et ces erreurs se passent dans certaines exemples, pas touts les exemples. On a déjà présenté cette limite dans le rapport précédent.