Documentation pour le développeur

Introduction :

On doit simuler une machine à pile. Pour cela, on procède en deux étapes.

Etape 1 - L'assembleur :

D'un fichier texte contenant un programme assembleur, on le traduira en langage machine dans un autre fichier.

Etape 2 - Simulation de la machine :

On simule ensuite la machine à pile. Pour cela, on récupère le fichier en langage machine et on l'exécute instruction par instruction.

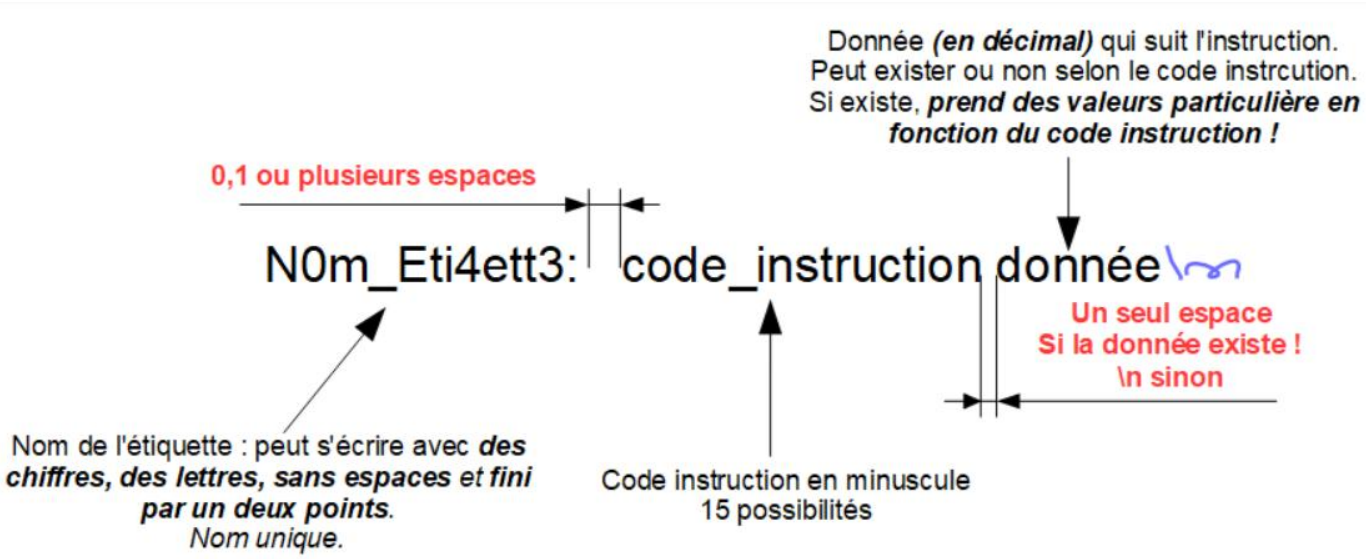
I/L'assembleur:

On sait que « l'exécutable prendra le nom du fichier à tester en argument ». C'est donc à partir de ce fichier que nous allons travailler. Appelons pour la suite fichier_assembleur.

On doit donc connaître comment ce fichier est organisé, comment est-il écrit ? Quelle est sa structure ? De ce fichier, on générera un autre qui correspondra à la traduction machine, qui lui possède aussi une certaine structure, que l'on devra respecter.

A/Traduction

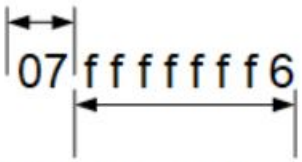
Supposons d'abord que le fichier en entrée est correct et ne comprend pas d'erreur de syntaxe. La structure générale de chaque ligne est donc la suivante :



De ce schéma, on construira nos fonctions qui permettront de traduire fichier_assembleur en un autre fichier contenant le langage machine. Appelons le fichier_machine.

Fichier_machine possède aussi une structure précise, que voici :

Code Instruction unique entre 0 et 14
écrit en ***hexadecimal*** ***(sur 2 octets).***



Donnée correspondante au code Instruction écrit en ***hexadecimal*** *sur* ***4 octets***.
Cette donnée numérique finale provient :

- D'une donnée numérique de fichier_assembleur
- D'une étiquette
- Si pas de donnée après le code instruction, on mettra 00000000 par défaut.

On récupérera donc dans l'ordre toutes les étiquettes, puis les codes instructions et enfin les données, s'il y en a. On utilise alors les fonctions :

```
int recuperation_etiquette(char* nom_fichier,int nombre_ligne,char* tab_etiquette[])
int traduction_instruction_octect_poids_fort(char* nom_fichier,int nombre_ligne,int *tab_instruction_courante_decimale)
int recuperation_donnee(char* nom_fichier,int nombre_ligne,int *tab_instruction_courante_decimale,char* tab_etiquette[],int *tab_donnee)
```

A.1/Recuperation étiquette(s)

Supposons que fichier_assembleur correspond à celui ci-dessous :

Ligne	
0	ici: read 1000
1	push 1000
2	push# 0
3	op 6
4	jpz fin
5	push 1000
6	op 5
7	pop 1000
8	write 1000
9	jmp ici
10	fin: halt

Pour chaque ligne i, s'il y a une étiquette on remplira le tableau tab_etiquette à l'indice i par l'étiquette correspondante. Sinon, on affectera une chaîne vide.
Reste maintenant à détecter les étiquettes.
Il y a une étiquette si est seulement si le caractère « : » est présent. Tout ce qui se trouve avant correspond à l'étiquette. On suppose également que l'étiquette ne fait pas plus de

cent caractères.

A.2/Recuperation code instruction

On initialise en variable globale le tableau ***Instruction*** tel que *pour chaque i∈{0,...,14} Instruction[i]* correspond au *code instruction numéro i.*

```
Instruction[]={"push","push#","ipush","pop","ipop","dup","op","jmp","jpz","rnd","read","write","call","ret","halt"};
```

On utilisera ce tableau, pour nous faciliter la tâche dans la recherche des instructions et dans leur traduction en nombre dans fichier_machine. Attention, à ne pas le faire naïvement...

Dans ***char ligne[nombre_caractere_max_ligne]*** on récupère la ligne. (On suppose que le nombre de caractère maximal par ligne ne dépasse pas 1000.) On vérifie ensuite si pour chaque i∈ {0,...,14}, Instruction[i] est présent ou pas dans ligne[], mais il faut le faire dans le bon ordre et ne pas le faire dans l'ordre croissant ! En effet, si on commence par chercher, est-ce que push est dans la ligne, on aura une réponse affirmative même si la ligne contient push# ! Certaines instructions sont les sous-chaînes d'autres instructions ! On commence donc par les instructions les plus petites et étant sous-chaîne d'autre instruction.
On conserve par la suite, dans ***tab_instruction_courante*** les codes instructions, l'indice i du tableau correspondant à la ligne i du fichier.

A.3/Recuperation données

Après avoir récupèré les étiquettes et les codes instructions, on récupèrera les données. Dans le tableau ***tab_donnee***, on va récupérer, l'ensemble des données nécessaire pour pouvoir enfin écrire le fichier en langage machine. On s'aidera bien sur de ***tab_instruction_courante*** et ***tab_etiquette***.
Il y a 3 cas à considérer :

1. Dans la ligne, il y a une instruction, qui ne nécessite pas de donnée : on mettra la valeur 0 dans ***tab_donnee***.
2. Dans la ligne, il y a une instruction, qui nécessite une donnée : on récupère la donnée et on la met dans le tableau ***tab_donnee***.
3. Dans la ligne, il y a une instruction, qui nécessite le traitement des étiquettes : on récupère la valeur qu'il faut et on la met dans le tableau ***tab_donnee***.

Quelque subtilité à noter:

Concernant le cas 2, une étiquette pouvant posséder les caractères numériques, on doit être sûr de dépasser le carctères « : », avant de commencer à stocker la donnée.

Concernant le cas 3, on doit être capable de récupèrer l'étiquette après l'instruction, que l'on comparera ensuite avec une des valeurs de ***tab_etiquette***. Etant donné l'unicité de l'étiquette, on a la formule suivante :

$$tab\_donnee[i]=j-i-1$$

avec i la ligne où l'étiquette est une donnée,

avec j la ligne ou l'étiquette est l'endroit de la rupture de séquence.
Cette valeur sera rajoutée au registre PC qui s'occupe d'informer le processeur sur la proche instruction à lire.

Un cas 4 est présent dans notre code. On en parlera à la fin de cette partie.
On peut à présent passer à la traduction, et écrire fichier_machine. On a toutes les informations utiles stockées dans ***tab_instruction_courante_decimale, tab_etiquette*** et ***tab_donnee.***

A.4/ Contraintes

Cette méthode nous a poussé à supposer qu'il n'y a pas de ligne vide dans fichier_assembleur. Si ligne vide il y a, un message d'erreur l'indiquera. Par ligne vide, on entend, tout saut de ligne inutile, ou saut de ligne contenant des espaces et tabulations.
On impose également que la fin du fichier, soit une ligne d'instruction, suivit d'un caractère saut de ligne ou pas.
Si ce caractère n'est pas présent, on le rajoute. On a fait ce choix, car on avait quelque doute sur la fonction ***fgets*** qui récupère les lignes. Puisqu'elle récupere chaque ligne grâce à la présence de '\n', récupère-t-elle la dernière ligne si '\n' n'est pas présent ?
Pour évacuer tous les doutes et faciliter ensuite la traduction, on a décidé de le rajouter à la fin, s'il n'est pas présent, et de faire un message d'erreur si des sauts de lignes en trop sont faits à la fin.

En somme, fichier_assembleur, ne doit pas posséder de ligne vide, et de saut de ligne inutile à la fin.

A.5/Outils

On utilise d'autres fonctions lors de la traduction, comme par exemple :

- int verif_fichier_vide(char* nom_fichier)
- int actualisation_fichier_code_assembleur(char* nom_fichier)
- int nombre_ligne_fichier(char* nom_fichier)

La dernière fonction permet de compter le nombre de ligne. Pour être plus exacte, elle compte le nombre de fois où le caractère '\n' apparaît. Si la dernière ligne ne comporte pas ce caractère, on peut oublier de la traiter, d'où l'ajout du caractère '\n' à la fin du fichier par la fonction ***actualisation_fichier_code_assembleur.*** Sécurité en plus, on mettra un cas numéro 4, lors de la récupération des données. Si une ligne ne possède aucune instruction (donc à la fin), on mettra la valeur 100 dans le tableau, valeurs ne correspondant à aucun code instruction.

B/Gestion d'erreur

On a supposé au départ que fichier_assembleur était correct. Comment a-t-on gérer les erreurs de syntaxe ?

Sachant que l’on doit renvoyer un message d’erreur en cas d'incohérence dans le

programme assembleur, on peut dégager plusieurs problématiques :

- On ne peut pas avoir plusieurs étiquettes avec le même nom !
- Vérifier que les codes instructions sont existants (parmi les 15 connus)
- Selon chaque instruction, on peut avoir ou non une donnée.
- S'il y en a une, on ne peut pas mettre n'importe laquelle.
- Il y a qu'une seule donnée à mettre en argument ! (quand c'est le cas)

Le tableau suivant résume les trois derniers points.

Code Instruction	**Y-a t-il des données à saisir ?** *(décimale)*	**Si oui, quelles sont leurs valeurs possibles ?**
push x	**Oui** Une adresse (sur la pile)	**[|0;5000|]**
pop x	**Oui** Une adresse (sur la pile)	**[|0;5000|]**
read x	**Oui** Une adresse (sur la pile)	**[|0;5000|]**
write x	**Oui** Une adresse (sur la pile)	**[|0;5000|]**
op x	**Oui** Une instruction prédéfinie	**[|0;15|]**
push# i	**Oui** Une Valeur entière	On veut que la donnée soit sur **4 octets** (Voir énoncé). On travaille avec des nombres signés **[|-2147483648;2147483647|]**

rnd x	**Oui**	On travaille sur 4 octets signés. Avec cette commande on génère une valeur aléatoire entre 0 et x-1. La valeur max pouvant être prise est donc x-1. Or x-1 ne peut pas dépasser le max de la plage d'un nombre sur 4 octets signés. Donc x ne doit pas dépasser le max de la plage d'un nombre sur 4 octets signés plus un. On commencera à 1 : si x vaut zéro l'intervalle n'est pas bien défini. **[|1;2147483648|]**
jmp adr	**Oui** Combien de sauts on effectue dans la lecture du programme machine. (Registre PC)	Étiquette existante
jpz adr	**Oui** Combien de sauts on effectue dans la lecture du programme machine. (Registre PC)	Étiquette existante
call adr	**Oui** Combien de sauts on effectue dans la lecture du programme machine. (Registre PC)	Étiquette existante
ipush	**Non**	
ipop	**Non**	
dup	**Non**	
ret	**Non**	
halt	**Non**	

B.1/Gestion d'erreur etiquettes

A cause de la syntaxe imposée à chaque ligne de fichier_assembleur, on commence d'abord par chercher les potentielles étiquettes, que l'on conserve dans ***tab_etiquette***.

Grâce à la fonction ***int verif_etiquette***, que chaque étiquette ne comporte pas d'espaces et on vérifie l'unicité de celle-ci.

B.1/Gestion d'erreur instructions et donnée

Si toutes les règles concernant les étiquettes sont bien respectées, on continue avec la vérification des instructions et des données associées. On s'aidera du tableau construit plus haut.

On sait que pour chaque ligne, la suite de caractères est précise, il y a un certain ordre logique. Cet ordre logique est-il respecte ?

Lorsqu'il n'y a pas d'étiquette, on a dans l'ordre :
1. des espaces et/ou des tabulations
2. une instruction

3_1. un saut de ligne, si l'instruction ne prend pas de paramètre
3_1bis. un saut de ligne, si l'instruction prend un paramètre et que ce parametre vaut 0. (Abscence de paramètre = 0 par defaut)
3_2: un espace, si l'instruction prend un paramètre

4. Si paramètre il y a, il faut que celui-ci soit cohérent. On finit par un saut de ligne.

Lorsqu'il y a une étiquette, on a dans l'ordre :
1. Tous ce qui est avant le caractere ':', correspond a l'étiquette
2. des espaces et/ou des tabulations
3. une instruction

4_1. un saut de ligne, si l'instruction ne prend pas de paramètre
4_1bis. un saut de ligne, si l'instruction prend un paramètre et que ce parametre vaut 0.
4_2. un espace, si l'instruction prend un paramètre

5. Si paramètre il y a, il faut que celui-ci soit cohérent. On finit par un saut de ligne.

Comment fonctionne notre vérification : on vérifie que toutes ces étapes sont respecter, avec des points de contrôle. On passera par une série de test, verifiant les différents ordres logiques évoqués.

Si l'un d'entre eux n'est pas vérifié, on renvoie le code erreur associé :

- Renvoie [|0;nombre de ligne-1|] si instruction inexistante.
- Renvoie [|nombre de ligne ; 2*nombre de ligne-1|] si erreur dans un paramètre d'une instruction.
- Renvoie [|4*nombre de ligne ; 4*nombre de ligne-1|] si il n'y a qu'une etiquette sur la ligne,sans instruction (donc ligne incomplete).
- Renvoie [|3*nombre de ligne ; 3*nombre de ligne-1|] si tabulation entre

instruction et le paramètre.
- Renvoie [|5*nombre de ligne ; 5*nombre de ligne-1|] ligne vide ou saut de ligne.

C/Ameliorations

Lors de la traduction, plusieurs étapes peuvent être faites dans une seul même fonction. On parcourt le fichier plusieurs fois, pour d'abord récupérer les étiquettes, puis, récupérer les instructions et enfin les données. On pourrait peut-être essayer de faire toutes ces étapes en même temps.

Conserver les différentes lignes dans une liste chaînes serait aussi une bonne idée.

On peut également essayer d'ignorer les lignes vides et sauts de lignes. On restreint sinon beaucoup trop la syntaxe.

II/La Simulation de la Machine:

Après avoir traduit le programme en langage machine, cette partie est dédiée à simulation du programme et à l'exécution de chaque instruction. Pour y parvenir, il a été nécessaire d'implémenter chaque instruction, après avoir initialisé la pile et récupérer toutes les instructions et les données.

A/ Récupération des lignes du programme et initialisation de la pile

Afin de pouvoir simuler les programmes, la fonction ***recuperation_instruction_donnee_fichier_hexa*** permet de récupérer linéairement chaque ligne du programme, et de stocker les instructions et les données dans les tableaux ***instruction_courante*** et ***donnée_courante***.

Par la suite, la structure de pile est initialisée par la fonction ***initialisation_pile*** telle que :

- Le registre ***PC*** qui stocke l'adresse de la prochaine instruction à exécuter est initialisé à 0.
- De même, le registre ***SP*** correspondant à l'adresse du sommet de la pile est initialisé à 0 car la pile est vide au démarrage.
- Tous l'espace de travail, c'est-à-dire les 5000 adresses, prendront la valeur 0. Cela correspond à une augmentation de la complexité, mais permet notamment de connaitre précisément où se situe le sommet de la pile, et quelles adresses de la mémoire sont occupées.

La pile pourra être affichée par l'utilisateur grâce à la fonction ***AfficherPile.***

B/Implémentation des instructions

Les instructions vont globalement permettre d'ajouter et de retirer des éléments à la pile, tout en ayant la possibilité de faire des opérations basiques sur la valeur de ses éléments. L'implémentation a été sans réelles difficultés pour ses fonctions car le sujet était très précis.
Après leurs implémentations, nous étions capables d'exécuter le programme à l'aide de la fonction ***execution_principale,*** prenant en arguments les tableaux ***instruction_courante*** et ***donnée_courante***, la pile, ainsi que le nombre de lignes.

Fonctionnement de la fonction ***execution_principale :***

On parcourt linéairement et de manière simultanée les tableaux ***instruction_courante*** et ***donnée_courante*** tant qu'il reste une instruction à exécuter. Puis, on identifie l'instruction grâce à son **code unique compris entre 0 et 14**. Après identification, on exécute la fonction implémentant celle-ci et on incrémente le ***registre PC***.