Projet d'Architecture des ordinateurs
Documentation pour le développeur

Tom PIETTON
Anas IBOUBKAREN

Février 2023

Sommaire

1 Introduction

Nous allons dans un premier temps présenter l'implémentation en justifiant nos choix. Puis, nous détaillerons les difficultés rencontrées au cours du projet et les éventuelles améliorations possibles.

2 Implémentation

2.1 Structures

Dans notre programme, nous utilisons trois structures :

- `etiq` pour récupérer les étiquettes
- `instruction` pour gérer les intructions
- `instructionNumerique` pour gérer les instructions traduites en code numérique

Une variable de type `etiq` possède deux champs : un champ `eti` qui est le nom de l'étiquette et un champ `ligne` qui correspond au numéro de la ligne où se trouve l'étiquette. La création de cette structure est motivée par le besoin de lier ces deux données pour l'affichage des erreurs ainsi que pour la traduction en code machine.

Une variable de type `instruction` possède deux champs : un champ `code` qui est la chaîne de caractères du mot-clef associé à l'instruction et un champ `donnee` qui correspond à la chaîne de caractères de la donnée associée à l'instruction. Cela nous permet, au fil du programme, de scinder chaque ligne entre un bloc mot-clef et un bloc donnée. La détection des éventuelles erreurs ainsi que la traduction en code numérique s'avèrent alors plus aisées.

Une variable de type `instructionNumerique` possède deux champs : un champ `code` qui est le numéro correspondant au mot-clef de l'instruction — un entier — et un champ `donnee` qui correspond à la donnée de l'instruction — un entier. La structure `instructionNumerique` est le symétrique du type `instruction`. Elle permet de faciliter le processus de traduction et la génération du fichier `hexa.txt`.

2.2 Récupération des étiquettes et des instructions

Dans un premier temps, nous extrayons du fichier source les *étiquettes* et les *instructions*. Les étiquettes sont stockées dans un tableau de pointeurs sur des variables de type `etiq` tandis que les instructions sont stockées dans un tableau de chaînes de caractères grâce à la fonction `recupEtiqLigne`. Cette dernière parcourt les lignes et fait appel à deux fonctions pour les traiter — `verifEtiquette` et `ligneTableauFichier`. La première fonction vérifie qu'il y a bien une étiquette par la détection des deux-points et, le cas échéant, récupère l'étiquette et la stocke dans le tableau `ETI` avec le numéro de ligne associé.

Lorsque nous récupérons les étiquettes, nous vérifions qu'elles sont toutes valides — *i.e.* elles sont composées uniquement de caractères alphanumériques — et qu'il ne figure aucune espace avant les deux-points. Après l'exécution de cette fonction, `doublonEtiq` permet non seulement de détecter les doublons parmi les étiquettes mais aussi de signaler les éventuelles erreurs à l'utilisateur (*cf.* section 2.5.7).

Pour les lignes, nous récupérons dans une chaîne de caractères tout ce qui suit le premier caractère qui est différent d'un caractère d'espacement — espace ou tabulation — depuis le début de la ligne ou des deux-points s'il y a une étiquette.

Expliquons le procédé avec l'exemple suivant :

```
ici:  read 1000
      push 1000
      push# 0
      op 6
      jpz fin
      push 1000
      op 5
      pop 1000
      write 1000
      jmp ici
fin:  halt
```

D'une part, nous obtenons le tableau suivant :

```
Etiq = [("ici", 1), ("fin", 11)]
```

où `(nomEtiquette, numLigne)` représente un élement de la structure `etiq`.

D'autre part, nous obtenons le tableau de chaînes de caractères suivant :

```
Fichier = ["read 1000",
           "push 1000",
           "push# 0",
           "op 6",
           "jpz fin",
           "push 1000",
           "op 5",
           "pop 1000",
           "write 1000",
           "jmp ici",
           "halt"]
```

Par la suite, nous vérifions qu'il n'y a *aucun doublon* parmi les étiquettes.
Nous complétons ensuite un tableau de pointeurs sur des variables de type `instruction` grâce à la fonction `creationInstructions` dans laquelle nous vérifions simplement que chaque ligne est de la forme :

- `bloc1 bloc2` s'il y a un *mot-clef* et une *donnée*
- `bloc1` s'il y a uniquement un *mot-clef*.

La cohérence des instructions n'est pas encore vérifiée. Nous obtenons alors le tableau suivant :

```
tab = [("read", "1000"),
       ("push", "1000"),
       ("push#", "0"),
       ("op", "6"),
       ("jpz", "fin"),
       ("push", "1000"),
       ("op", "5"),
       ("pop", "1000"),
       ("write", "1000"),
       ("jmp", "ici"),
       ("halt", "@")]
```

Il est à noter que l'absence de donnée est encodée par une chaîne contenant uniquement le caractère `@`. C'est un choix totalement arbitraire effectué dans l'objectif de clarifier le code.

Il nous faut désormais contrôler la cohérence des mots-clefs et des données. Nous avons décidé de traiter cette étape en même temps que le remplissage d'un tableau de pointeurs sur la structure `instructionNumerique` à l'aide de la fonction `coherenceInstructions`.

Nous vérifions tout d'abord que le mot-clef de chaque ligne correspond à un mot-clef du langage assembleur. Puis nous nous assurons que la donnée correspondante au mot-clef est cohérente — dans les bons intervalles pour une donnée numérique et référant à une étiquette pour les chaînes de caractères. Pour ce faire, nous avons classé les *mots-clefs* en trois catégories :

- Le `type1` correspond aux mots-clefs attendant une donnée numérique.
- Le `type2` correspond aux mots-clefs attendant une étiquette ou une donnée numérique.
- Le `type3` correspond aux mots-clefs n'attendant aucun argument.

Nous avons deux fonctions gérant les deux premiers types nommés `coherenceLigneType1` et `coherenceLigneType2`. Ces fonctions récupèrent la donnée qui est une chaîne de caractères et vérifient sa cohérence avec le mot-clef la précédant grâce à des bornes **min**/**max** ou à un **tableau d'étiquettes**. Elles permettent donc le signalement de toutes les erreurs relatives à la syntaxe des mots-clefs ainsi qu'à la cohérence et à la syntaxe des données (*cf.* section 2.4). Le troisième type, moins complexe, est géré directement à l'intérieur de la fonction.

2.3 Traduction

Si l'étape précedente ne conduit pas à l'exécution de l'instruction `exit(1)`, il n'y a *aucune erreur* et nous pouvons créer un fichier `hexa.txt`. À l'aide du tableau rempli à l'étape précédente, nous effectuons une traduction en recopiant les instructions numériques en hexadécimal. Le fichier `hexa.txt` contient ainsi :

```
0a 000003e8
00 000003e8
01 00000000
06 00000006
08 00000005
00 000003e8
06 00000005
03 000003e8
0b 000003e8
07 fffffff6
0e 00000000
```

2.4 Exécution

Dans un second temps, nous abordons la phase *distincte* de simulation du programme. Afin d'exécuter le programme, nous récupérons son contenu dans un tableau de pointeurs sur des données de type `instructionNumerique` en parcourant le fichier `hexa.txt`. Ce tableau est donc identique à celui créé pour effectuer l'étape de traduction sur le fichier `hexa.txt`.

Par la suite, nous simulons la mémoire grâce à une déclaration statique d'un tableau `memoire` de 5000 entiers. Puis nous lançons l'exécution grâce à la fonction `execution`. Celle-ci initialise

un pointeur de pile `SP` et un registre `PC` dans lequel est stocké l'indice de la prochaine instruction à exécuter. Lorsque nous exécutons le programme, nous passons par adresse ces deux entiers dans chacune des fonctions encodant une instruction afin qu'ils puissent subir des modifications. Les erreurs d'exécutions sont détaillés dans la section 3.3.

2.5 Affichage des erreurs de compilation

Lors de la compilation, nous repérerons tous les problèmes pouvant empêcher de générer le fichier `hexa.txt` et les signalons à l'utilisateur. Afin de faciliter l'utilisation de notre programme, nous avons préféré afficher les erreurs par thématiques afin que l'utilisateur n'ait pas à réexécuter notre code constamment.

Dans un premier temps, nous vérifions qu'un fichier est en entrée et, le cas échéant, que celui-ci ne soit pas vide (*cf.* sections 2.5.1 et 2.5.2).

Ensuite, à l'aide des fonctions `recupEtiqLigne` et `doublonEtiq`, nous affichons les erreurs concernant les lignes sans instruction – vides ou avec uniquement une étiquette –, les syntaxes d'étiquettes invalides et les doublons parmi celles-ci (*cf.* sections 2.5.3, 2.5.4 et 2.5.7).

Puis, nous identifions avec la fonction `creationInstructions` si chaque ligne est de *structure correcte* – *cf.* section 2.2 (*cf.* section 2.5.8 pour le code d'erreur associé).

Finalement, à l'aide de la fonction `coherenceInstructions`, nous détectons la validité de la syntaxe des mots-clés et des données mais aussi leur cohérence (*cf.* sections 2.5.4, 2.5.6, 2.5.9, 2.5.10 et 2.5.11).

Afin d'afficher les erreurs, nous utilisons de la couleur pour gagner en *clarté*. Chaque message d'erreur est précédé de `Erreur :` et les zones détéctées comme étant responsables de l'erreur sont en `violet`. Cela permet d'aider l'utilisateur à mieux corriger son code.

Voici la liste des messages d'erreur liée à la compilation.

2.5.1 Aucun fichier en entrée

```
Erreur : Il n'y a pas de fichier en entrée
```

2.5.2 Fichier vide

```
Erreur : Fichier vide
```

2.5.3 Ligne vide

```
Erreur : Absence d'instruction dans le code Assembleur à la ligne 2
```

2.5.4 Étiquette invalide

```
Erreur : Nom d'étiquette non valide à la ligne 1
eti!:  push 1000
```

2.5.5 Mot-clef invalide

```
Erreur : Mot-clef invalide à la ligne 1
pushh 1000
```

2.5.6 Donnée invalide

```
Erreur : La donnée numérique à la ligne 1 dépasse l'intervalle autorisé [0;4999]
push 6000
```

2.5.7 Doublon d'étiquettes

```
Erreur : Il y a des doubles parmi vos étiquettes aux lignes 1 et 2
loop: read 1000
loop: read 2000
```

2.5.8 Bloc non conforme

```
Erreur : Bloc donnée non conforme à la ligne 1
read 10 00
```

2.5.9 Donnée non numérique

```
Erreur : La donnée à la ligne 1 n'est pas un nombre
read 10k
```

2.5.10 Étiquette inexistante

```
Erreur : Etiquette introuvable à la ligne 1
jmp loop
```

2.5.11 Présence de donnée

```
Erreur : Donnée non attendue à la ligne 1
halt bonjour
```

3 Difficultés rencontrées

Dans cette troisième partie, nous aborderons les problèmes rencontrés et expliquerons la réflexion qui nous a permis de les résoudre.

3.1 Gestion de la détection des erreurs

Afin de ne pas alourdir ce document et cette sous-section, nous ne détaillerons pas la manière dont nous avons traité les erreurs – cela étant fait à la section 2.5. Nous tenons cependant à signaler que l'ordre dans lequel nous les gérons et les affichons a été le fruit d'une réflexion et de multiples essais. En effet, lister toutes les erreurs lors de la première exécution était notre premier objectif et nous l'avons réalisé. Cependant, il s'est avéré que l'affichage était peu lisible et qu'une telle gestion des erreurs ne s'accordait pas bien avec notre implémentation. Afin de solutionner ce problème, nous avons préféré scinder l'affichage de sorte à ce qu'il reste cohérent et qu'un nombre maximal d'erreurs soient signalées proprement par des codes d'erreur clairs et pertinents.

3.2 Gestion de la mémoire

Nous avons été confrontés à la difficulté d'une libération correcte de la mémoire allouée grâce aux fonctions `malloc` et `realloc`. En effet, si nous arrêtons le programme directement après une suite d'erreurs, il ne faut pas continuer à exécuter le code. Mais si nous nous contentons d'exécuter `exit(1)`, nous ne libérerons pas proprement l'espace mémoire alloué. Pour résoudre ce problème, nous avons décidé d'introduire une variable dans le `main` nommée `condArret` que nous passons par adressse dans toutes nos fonctions et qui permet de détecter la présence des erreurs. En exploitant la valeur de cette variable avec l'instruction `goto` – qui aurait pu ne pas être utilisée mais qui économise un peu d'espace dans le code du `main` –, nous égalisons le nombre d'`alloc` et de `free` en toutes situations.

3.3 Gestion des erreurs d'exécution

Il faut finalement s'assurer que les instructions qui s'exécutent dans la fonction `execution` ne provoquent pas d'erreur sur le terminal. Nous avons donc essayé d'identifier toutes les failles pouvant survenir lors de l'étape d'exécution. Afin d'approcher — et nous l'espérons atteindre — l'exhaustivité, notre programme remonte les erreurs suivantes :

- Pour chaque opération nécessitant de récupérer plus d'une donnée (`ipush`, `ipop`, `op`, *etc.*), on vérifie à l'aide du pointeur de pile que nous ne sommes pas confrontés à une tentative d'accès mémoire non légitime – *i.e.* un accès à une zone mémoire hors du tableau `memoire`). Par exemple, cela risque d'arriver si le pointeur de pile est à 0 et que l'on essaie d'exécuter l'instruction `op 0`. Pour signaler ces erreurs, nous fixons le pointeur de pile à -1, ce qui conduit à l'exécution d'un bloc affichant un message d'erreur (*cf* section 3.3.2) puis à l'arrêt de l'exécution.

- Parmi les cas déjà précisés précédemment, nous relevons également les boucles infinies. Une boucle infinie est une *erreur de programmation*, nous n'interferons donc pas si le programmeur écrit en assembleur `jmp -1` – cela relève de sa responsabilité. Toutefois, certaines boucles infinies sont des erreurs du même type que celles signalées précédemment. En effet, toute boucle de la forme `jpz -1` conduit à un dépilement infinie de la pile et l'erreur est bien traitée par la vérification finale du pointeur de pile dans la fonction `execution`.

- De plus, la fonction `ret` peut s'avérer dangereuse si elle est mal utilisée. En effet, nous pouvons faire un retour de procédure et récupérer sur la pile une donnée qui n'est *absolument* pas valide et qui entraînerait une erreur d'accès mémoire dans le tableau où nous stockons les instructions du programme assembleur : `tab3`. Afin d'éviter ce type de problème, une vérification est effectuée à la fin de chaque `ret`. Si le registre `PC` possède une valeur supérieure ou égale au nombre de lignes du programme assembleur ou est strictement inférieur à 0 – *i.e.* lors de la prochaine réalisation de la boucle d'exécution, un accès mémoire invalide se produira, le programme s'arrête pour cause de tentative d'accès à une instruction inexistante. Une erreur est donc signalée à l'utilisateur avant la fermeture du programme (*cf* section 3.3.1).

- Toute tentative de division par 0 (via l'opérateur `/` ou `%`) est interdite et entraîne une fin d'exécution accompagnée d'un message d'erreur (*cf* section 3.3.3). Cela se fait en donnant arbitrairement au registre `SP` la valeur `-100`.

- Finalement, nous traitons le cas de la pile pleine. Nous considérons que la pile peut occuper toute la mémoire et qu'une pile pleine est signalée par l'égalité de `SP` à 5000.

Dans ce cas, le programme ne s'arrête évidemment pas mais tout appel aux fonctions : `dup`, `push`, `push#` et `rnd` entraîne une erreur (*cf.* section 3.3.4) de *stack overflow* et conduit à la fermeture du programme.

3.3.1 Instruction inexistante

`Erreur : Tentative d'accès à une instruction inexistante`

3.3.2 Accès mémoire interdit

`Erreur : Tentative d'accès mémoire interdite`

3.3.3 Tentative de division par 0

`Erreur : Tentative de division par 0`

3.3.4 Pile pleine

`Erreur : La pile est pleine`

4 Améliorations possibles

4.1 Liste chaînée ?

Nous pensons qu'une amélioration possible de notre programme pourrait être l'implémentation d'une structure de liste chaînée permettant de réduire le nombre de structures utilisées afin de traiter le code assembleur. Cependant, nous ne sommes pas entièrement convaincu que cela optimise considérablement notre code. En effet, nous n'avons pas besoin de supprimer certains éléments au fil du programmme. De plus, pour centraliser les structures il faudrait équiper chaque maillon d'un champ étiquette et cela peut prendre inutilement de la place puisque nous allons stocker le pointeur `NULL` dans un grand nombre de maillons. Toutefois, cela permettrait de rendre notre code plus lisible et éventuellement de faciliter certaines étapes comme la libération de la mémoire.

4.2 Légèreté du code

La seconde partie du code portant sur l'exécution contient parfois quelques répétitions qu'il nous semble possible d'éviter assez facilement. Nous pensons qu'il serait possible de réduire le nombre de lignes sans perdre en clarté en utilisant la fonction `switch` ou des blocs conditionnels plus astucieux.

4.3 Affichage des erreurs

Nous sommes totalement satisfaits par la manière dont les erreurs s'affichent. Cependant, il serait possible de les coupler avec certains `warning`. Par exemple, si l'utilisateur utilise `ret` sans utiliser `call`, il pourrait être judicieux de lui signaler de faire attention puisque cela risque de le conduire à une erreur ou à un résultat non souhaité.

Afin de traiter les erreurs d'étiquettes invalides, nous pouvons aussi détecter les légères erreurs de recopie avec la distance de Levenshtein. En effet, c'est une distance comptant le nombre minimal d'édition qu'il faut effectuer pour passer d'une chaîne à une autre. Il est donc facile de parcourir les étiquettes du début et d'afficher une proposition de remplacement dès lors que la

distance de Levenshtein est inférieure ou égale à 2 par exemple. Nous n'avons pas implémenté ces fonctionnalités puisque nous trouvons qu'elles sortaient légèrement du cadre du projet mais nous souhaitons malgré cela relever leur pertinence.