Module EISE4 : Compilation

TPs 1 & 2 : Machine virtuelle, Assembleur, Structures de données de manipulation d'arbres

Ce premier TP est consacré à l'écriture en langage C de trois modules importants de la chaîne de compilation : la machine virtuelle, l'assembleur, et les fonctions C consacrées à la création et au parcours d'arborescences que sont les ASA (Arbres de Syntaxe Abstraite, Abstract Syntax Tree ou AST en anglais). Chaque module fait l'objet d'une partie de ce long TP nécessaire pour disposer de toute la partie « back-end » du compilateur. En annexe se trouve la liste de toutes les instructions reconnues par la machine virtuelle EVM (EISE Virtual Machine).

PARTIE 1 : La machine virtuelle EVM

La machine virtuelle **evm**, dont vous écrirez le fichier source dans **evm.c**, prend un fichier langage machine en paramètre, l'ouvre, et exécute les instructions qui s'y trouvent. Ce fichier langage machine, pour la lisibilité, est en fait un fichier texte qui a la structure suivante :

```
nombre_total_instructions
0 : instruction0
1 : instruction1
…
nombre_total_instructions-1 : instruction nombre_total_instructions-1
```

Ainsi, le fichier **e1.bin** suivant :

```
10
0:100
1:4
2:100
3:5
4:3
5:100
6:2
7:1
8:301
9:403
```

contient un code langage machine qui fait 10 instructions, et qui occupe les adresses 0 à 9. Aux adresses 0 et 1, vous pourrez reconnaître l'instruction **push 4**, puis **push 5** aux adresses 2 et 3. A l'adresse 4 une instruction **mult**, un nouveau **push 2** aux adresses 5 et 6, une instruction **add** à l'adresse 7, une instruction **output** en 8 et l'instruction d'arrêt de la machine virtuelle **halt** à l'adresse 9. L'exécution de ce code machine doit afficher la valeur 22.0 à l'écran. Une des fonctions vous allez écrire, **readAssembly(FILE *fin)**, doit lire le contenu de ce fichier et stocker les entiers correspondants aux instructions dans le tableau **codeSegment**.

Les 7 premières questions du TP se trouvent dans le fichier **evm.c** fourni.

Les suivantes sont :

Q8) Pour tester votre EVM, Décrypter le fichier **e2.bin** fourni suivant :

```
29
0:302
1:105
2:110
3:112
4:117
5:116
6:32
7:112
8:117
9:105
10:115
11:32
12:111
13:117
14:116
15:112
16:117
17:116
18:0
19:14
20:1
21:100
22:0
23:300
24:100
25:0
26:104
27:301
28:403
```

puis lancer votre machine virtuelle et observer le résultat. Que fait ce programme en langage machine ? Si besoin, dessiner les mouvements de pile, instruction par instruction.

Q9) Ecrire « à la main » votre propre fichier **e3.bin** permettant de tester la bonne exécution d'une majorité des instructions de l'EVM.

PARTIE 2 : L'assembleur ASM

L'assembleur **asm**, dont vous écrirez le fichier source dans **asm.c**, prend un fichier assembleur (avec extension de fichier **.asm**) comme argument d'entrée et fournit en sortie le fichier en langage machine (avec extension de fichier **.bin**) correspondant, directement compatible avec **evm**.

Un fichier assembleur contient une suite finie de lignes de texte, terminée par une dernière ligne contenant la pseudo instruction **end**. Quand la boucle de parcours de cette suite de lignes découvre la ligne contenant **end**, le code assembleur est considéré comme entièrement lu, et les deux passes d'assemblage proprement dites commencent.

L'assembleur que vous allez réaliser est un assembleur deux passes. La première va effectuer une traduction des instructions décrites sous forme de chaîne de caractères dans le fichier **.asm** (avec arguments ou sans) dans les entiers correspondants (avec arguments entiers ou pas, en parfaite correspondance avec le texte assembleur). Cette passe correspond à la génération du code machine

à partir du texte source assembleur. Si une instruction de rupture de séquence (un saut ou un call) fait référence à une étiquette qui n'a pas été encore rencontrée dans le texte assembleur, cette première passe prend note de l'endroit exact dans le code généré ou il manque des informations et ajoute une référence non résolue à la table des références non encore résolues. Si la ligne assembleur décodée est une déclaration d'étiquette (**etiq1:** par exemple), cette étiquette est déclarée (sans le caractère :) dans la table des étiquettes et sera exploitée dans la passe 2 de résolution des références inconnues.

Une fois la passe 1 terminée, le code généré possède sa taille finale, mais présente des « trous » pour chaque référence non résolue. Il suffit alors d'utiliser conjointement la table des étiquettes et la table des références pour remplacer ces trous par leurs valeurs exactes.

Exemple

Si l'on prend le code assembleur d'entrée suivant :

```
debut:
        inc     1
        push    0
        input
        push    0
        cont
        pushr   10.0
        gt
        jf      l1
        outchar ">10"
        halt
l1:
        outchar "<10"
        halt
        end
```

la première phase d'assemblage va produire le code langage machine suivant :

La ligne 9, qui contient une instruction de saut si faux **jf** vers une étiquette qui n'a pas encore été vue par l'assembleur puisqu'elle se trouve à la ligne 11, va se traduire par la génération du code suivant :

```
29
0:14
1:1
2:100
3:0
4:300
5:100
6:0
7:104
8:101
9:49
10:48
11:46
12:48
13:0
14:13
15:201
16:-1             // -1 car on ne connait pas encore l'adresse de l1
17:302
```

```
18:62
19:49
20:48
21:0
22:403
23:302
24:60
25:49
26:48
27:0
28:403
```

avec **201** qui représente le code machine de l'instruction à l'adresse 15, et **-1** qui représente la valeur entière (l'adresse où doit se faire le saut) correspondant à une adresse de branchement encore inconnue. En plus de ce code généré, on ajoute à la table des références une nouvelle entrée décrivant le fait qu'il y a une nouvelle référence inconnue vers une étiquette **l1** encore inconnue à l'adresse **16** dans le code.

Lors de la seconde passe, il suffit de parcourir en entier la table des références et, pour chaque entrée, à partir du nom de l'étiquette stockée, de retrouver celle-ci dans la table des étiquettes, de récupérer l'adresse absolue correspondante, et de stocker celle-ci à l'adresse correspondante dans le code (et donc d'écraser la valeur -1 mise précédemment de manière temporaire avec la bonne adresse).

Pour réaliser l'assembleur, on va avoir besoin de 4 structures de données :
- le segment de code, **codeSegment**
- la table des etiquettes, **tabLabels**
- la table des références, **tabReferences**
- la table des mots-clés représentant les instructions, **tabInstructionNames**

Les structures de données C correspondantes sont fournies, avec explications, dans le fichier **asm.c**. Il y a 11 questions dans ce fichier **asm.c**, qu'il faut traiter dans l'ordre.

Q12) Valider le bon fonctionnement de votre assembleur avec un fichier **.asm** testant l'ensemble des instructions.

PARTIE 3 : Fonctions de manipulation d'arborescences

Dans cette partie, il s'agit d'écrire des fonctions C permettant de créer et de parcourir des arborescences afin de générer du code assembleur que l'on pourra assembler et exécuter ensuite avec la machine virtuelle.

Avant d'écrire ces fonctions dans les fichiers **ast.h** et **ast.c**, il faut prendre le temps d'étudier et d'apprécier un outil génial disponible partout et notamment sur les machines de TP, l'outil **dot**. **dot** est un logiciel qui, à partir d'une description textuelle d'un graphe (orienté ou non), est capable d'en produire une représentation graphique, par exemple au format **png**. Par exemple, si l'on écrit le fichier **res.dot** suivant :

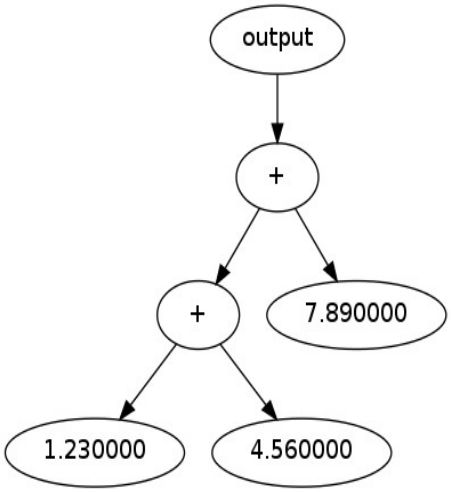
```
digraph {
        N000 [label="1.230000"]
        N001 [label="4.560000"]
        N002 [label="+"]
        N003 [label="7.890000"]
        N004 [label="+"]
        N005 [label="output"]
```

```
        N005 -> N004
        N004 -> N002
        N004 -> N003
        N002 -> N000
        N002 -> N001
}
```

et qu'on lance ensuite la commande :

```
dot -Tpng res.dot -o res.png
```

on obtient le fichier **res.png** suivant :



ce qui, vous l'admettrez, est super cool pour vite visualiser un AST. La syntaxe d'un tel fichier est simple. Après une première ligne disant qu'il s'agit d'un « directed graph » **digraph**, on définit dans un premier temps tous les nœuds de l'arborescence en les numérotant (nœuds **N000** à **N005**) et en leur associant un label entre **[]**. Dans un deuxième temps, on définit les arcs entre les nœuds avec des lignes comme **N004 -> N002**, qui signifie **N004** a pour fils **N002**.

Les structures de données C pour construire les arbres vont être amenées à évoluer, il faut les concevoir avec soin et penser fortement récursif pour écrire les fonctions correspondantes. Pour l'instant, nous allons considérer qu'il y a deux types de nœuds dans nos arborescences, des nœuds « numeric » représentant des valeurs flottantes, qui sont terminaux (ils n'ont pas de fils), et des nœuds de type operator qui pourront soit représenter des opérations arithmétiques (+,-,*,/) ayant deux fils (le fils 0 et le fils 1) ou l'opération « output ». De cette manière, on pourra créer un AST contenant des expressions arithmétiques faisant intervenir les opérateurs classiques et des constantes réelles, et l'opérateur « output » nous permettra d'afficher le résultat de cette expression à l'écran.

Les questions relatives à cette partie se trouvent dans les fichiers **ast.h** et **ast.c**. L'objectif affiché est de vous permettre, à partir d'un AST que vous aurez construit, de générer le code assembleur correspondant **.asm** et de l'exécuter avec l'EVM.

Annexe 1 : Les instructions assembleur de la EISE Virtual Machine

Les instructions arithmétiques et logiques effectuent des opérations de calcul sur des mots mémoire situés au sommet de la pile d'exécution. Elles ne nécessitent pas d'opérande car ceux-ci se trouvent déjà dans la partie haute de la pile (sous sommet et sommet de pile). Le compteur ordinal incrémenté d'une unité à la fin de chacune de ces instructions. Voici le détail de ces instructions :

ADD les deux opérandes au sommet de la pile sont additionnés et remplacés par le résultat de l'addition. Le pointeur de pile est décrémenté d'une unité.

SUB l'opérande au sommet de la pile est soustrait de l'opérande juste en dessous du sommet. Ces deux opérandes sont remplacés par le résultat de la soustraction. Le pointeur de pile décroît d'une unité.

MULT les deux opérandes du sommet de la pile sont multipliés et remplacés par le résultat de la multiplication. Le pointeur de pile décroît d'une unité.

DIV l'opérande juste en dessous du sommet de la pile est divisé par l'opérande du sommet de la pile. Ces deux opérandes sont remplacés par le résultat de la division. Le pointeur de pile décroît d'une unité. Dans le cas où le diviseur vaut 0, une erreur est détectée et l'exécution est arrêtée.

POW l'opérande juste en dessous du sommet de la pile est calculé à la puissance donnée par l'opérande du sommet de la pile. Une erreur d'exécution est provoquée en cas de calcul mathématique faux. Les deux opérandes du sommet de la pile sont remplacés par le résultat de l'opération. Le pointeur de pile décroît d'une unité.

NEG (NEGate) le signe de l'opérande au sommet de la pile est inversé. Le pointeur de pile reste inchangé.

AND une opération logique et est effectuée entre les deux opérandes du sommet de la pile. Le résultat de cette opération remplace les deux opérandes au sommet de la pile. Le pointeur de pile décroît d'une unité.

OR une opération logique ou est effectuée entre les deux opérandes du sommet de la pile. Le résultat de cette opération remplace les deux opérandes au sommet de la pile. Le pointeur de pile décroît d'une unité.

NOT l'opération logique non est effectuée sur l'opérande au sommet de la pile, c'est-à-dire que si l'opérande vaut faux, il devient vrai et inversement. La valeur 0 est interprétée comme faux, les autres valeurs comme vrai. Le pointeur de pile reste inchangé.

EQ (EQual) les deux opérandes au sommet de la pile sont dépilés et s'ils sont égaux, la valeur logique vrai 1 est placée sur le sommet de la pile, autrement c'est la valeur logique faux 0. Le pointeur de pile décroît d'une unité.

LS (LesS) les deux opérandes au sommet de la pile sont dépilés et si l'opérande juste en dessous du sommet est strictement inférieur à l'opérande du sommet, la valeur logique vrai 1 est placée sur le sommet de la pile autrement c'est la valeur logique faux 0. Le pointeur de pile décroît d'une unité.

GT (Greater Than) les deux opérandes au sommet de la pile sont dépilés et si l'opérande juste en dessous du sommet est strictement supérieur à l'opérande du sommet, la valeur logique vrai 1 est placée sur le sommet de la pile, autrement c'est la valeur logique faux 0. Le pointeur de pile décroît d'une unité.

Les instructions de manipulation de la pile effectuent des opérations sur les mots mémoire situés au sommet de la pile d'exécution. Voici le détail de ces instructions:

DEC op cette instruction décrémente le pointeur de pile SP de la valeur de l'opérande. Le compteur ordinal est incrémenté de 2.

INC op cette instruction incrémente le pointeur de pile SP de la valeur de l'opérande. Le compteur ordinal est incrémenté de 2.

PUSH op l'opérande est placé sur le sommet de la pile. Le pointeur de pile croît d'une unité. Le compteur ordinal est incrémenté de 2.

PUSHR op l'opérande est en fait constitué d'une suite de caractères terminée par le caractère 0. Cette suite représente un nombre réel qui est placé après décodage sur le sommet de la pile. Par conséquent, le pointeur de pile est incrémenté d'une unité et le compteur ordinal est incrémenté du nombre de chiffres constituant le nombre réel plus un.

LIBP op (Load Indexed by BP) l'opérande indique un déplacement qui est ajouté à la valeur du registre de base BP. La somme résultante est chargée sur le sommet de la pile. Le pointeur de pile croît d'une unité. Le compteur ordinal est incrémenté de 2.

DUPL la valeur au sommet de la pile est dupliquée au sommet de la pile. Le pointeur de pile et le compteur ordinal sont incrémentés de 1.

CONT le sommet de la pile est remplacé par l'élément conservé à l'adresse indiquée par la valeur située sur le sommet de la pile. Avant d'exécuter cette instruction, le sommet de la pile doit contenir l'adresse d'une cellule mémoire de la pile d'exécution. Le pointeur de pile et le compteur ordinal sont incrémentés de 1.

MOVE op l'opérande indique le nombre de mots mémoire qui doivent être enlevés et transférés du sommet de la pile vers une zone mémoire. L'adresse du début de cette zone mémoire doit être le premier mot mémoire en dessous des mots mémoire à transférer. Cette adresse ainsi que les mots mémoire à transférer sont dépilés. Le pointeur de pile est décrémenté de op+1 unités. Le compteur ordinal est incrémenté de 2.

COPY op le sommet de la pile est remplacé par un nombre de mots d'une zone mémoire dont l'adresse de départ est indiquée par le sommet de la pile. C'est l'opérande qui fixe le nombre de mots mémoire à copier. Au départ, le sommet de la pile doit contenir l'adresse de la zone de mémoire d'origine. C'est en fait une instruction **CONT** généralisée. Le pointeur de pile est incrémenté de op-1 unités. Le compteur ordinal est incrémenté de 2.

Les instructions de rupture de séquence permettent d'effectuer des branche-ments vers d'autres instructions du segment de code en changeant la valeur du compteur ordinal. Voici le détail des ces instructions :

JP op (JumP) effectue un branchement inconditionnel à l'adresse indiquée par l'opérande. Le compteur ordinal aura comme nouvelle valeur la valeur de l'opérande. Le pointeur de pile n'est pas affecté.

JF op (Jump if False) effectue un branchement à l'adresse indiquée par l'opérande sous condition que le sommet de la pile ait comme valeur logique faux (0). Si cette condition n'est pas vérifiée, le compteur ordinal est simplement incrémenté de 2 (on passe à l'instruction suivante). Dans tous les cas, cette valeur est dépilée et, par conséquent, le pointeur de pile est décrémenté d'une unité.

JL op (Jump if Less) effectue un branchement à l'adresse indiquée par l'opérande sous condition que la valeur juste en dessous du sommet de la pile soit strictement inférieure à la valeur du sommet de la pile. Dans tous les cas, ces deux valeurs sont dépilées et, par conséquent, le pointeur de pile est décrémenté de 2.

JG op (Jump if Greater) effectue un branchement à l'adresse indiquée par l'opérande sous condition

que la valeur juste en dessous du sommet de la pile soit strictement supérieure à la valeur du sommet de la pile. Dans tous les cas, ces deux valeurs sont dépilées et, par conséquent, le pointeur de pile est décrémenté de 2.

CALL op l'adresse de retour (PC+2) est placée sur le sommet de la pile d'exécution, puis il y a un branchement inconditionnel dans le code segment à l'adresse indiquée par l'opérande. Le pointeur de pile est incrémenté de 1 et le compteur ordinal a comme nouvelle valeur la valeur de l'opérande.

RET le sommet de la pile est dépilé et cette valeur est affectée au compteur ordinal. Le pointeur de pile est décrémenté de 1 et le compteur ordinal a comme nouvelle valeur la valeur qui se trouvait sur le sommet de la pile.

Les instructions d'entrée-sortie permettent à la machine virtuelle de communiquer avec le monde extérieur. Trois instructions sont prévues à cet effet :

IN le processeur attend que l'utilisateur ait entré un nombre au clavier de l'ordinateur. Ce nombre est stocké dans la case mémoire de la pile d'exécution à l'adresse indiquée par le sommet de la pile. Ensuite cette adresse est dépilée. Le pointeur de pile est décrémenté de 1 tandis que le compteur ordinal est incrémenté de 1.

OUT le sommet de la pile est considéré comme un nombre et est affiché à l'écran à l'endroit du curseur. Ensuite ce nombre est dépilé. Le pointeur de pile est décrémenté de 1 tandis que le compteur ordinal est incrémenté de 1.

OUTCHAR chaîne tous les mots mémoire qui suivent cette instruction dans le segment de code jusqu'au premier caractère 0 et forment une chaîne de caractères qui est affichée à l'écran à l'endroit du curseur. Le pointeur de pile n'est pas affecté. Le compteur ordinal est incrémenté de 1 plus le nombre de caractères formant la chaîne, 0 compris.

Enfin, il ne reste qu'à décrire les trois instructions spéciales de notre machine virtuelle.

SAVEBP le pointeur de base BP est sauvegardé sur la pile. La nouvelle valeur du pointeur de base est l'adresse du mot mémoire de la pile qui suit le mot mémoire dans lequel l'ancien BP vient d'être sauvegardé. Le pointeur de pile et le compteur ordinal sont tous deux incrémentés d'une unité.

RSTRBP (ReSToReBP) le sommet de la pile est dépilé et cette valeur est affectée au registre de base BP. Le pointeur de pile est décrémenté d'une unité tandis que le compteur ordinal est incrémenté d'une unité.

HALT provoque l'arrêt de la machine virtuelle et redonne le contrôle au système d'exploitation hôte.

La réalisation de l'interprète ne donne lieu à aucune complication particulière. Les trois registres PC, SP et BP sont initialisés, puis la boucle principale d'interprétation est lancée et ne s'arrête que si l'instruction courante est l'instruction HALT. Le compteur ordinal doit avoir comme valeur initiale l'adresse de la première instruction exécutable du programme, tandis que SP et BP sont initialisés à -1. Il est important de noter que le pointeur de pile SP indique le numéro de la cellule contenant le sommet de pile. L'incrémentation du pointeur de pile se fait avant l'affectation du sommet. Ainsi, la première action d'empilement se fera à l'adresse 0 (valeur initiale -1 plus 1 = 0).

Les opcodes du fichier `vm_codops.h`

```
#define OP_ADD          1
#define OP_SUB          2
#define OP_MULT         3
#define OP_DIV          4
#define OP_DIVI         5

#define OP_NEG          7
#define OP_AND          8
#define OP_OR           9
#define OP_NOT          10
#define OP_EQ           11
#define OP_LS           12
#define OP_GT           13

#define OP_INC          14
#define OP_DEC          15

#define OP_PUSH         100
#define OP_PUSHR        101
#define OP_LIBP         102
#define OP_DUPL         103
#define OP_CONT         104
#define OP_MOVE         105
#define OP_COPY         106

#define OP_JP           200
#define OP_JF           201
#define OP_JL           202
#define OP_JG           203
#define OP_CALL         204
#define OP_RET          205

#define OP_INPUT        300
#define OP_OUTPUT       301
#define OP_OUTCHAR      302

#define OP_CALLS        400
#define OP_SAVEBP       401
#define OP_RSTRBP       402
#define OP_HALT         403
```