

A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

09/06/2019

# Rapport projet Informatique

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

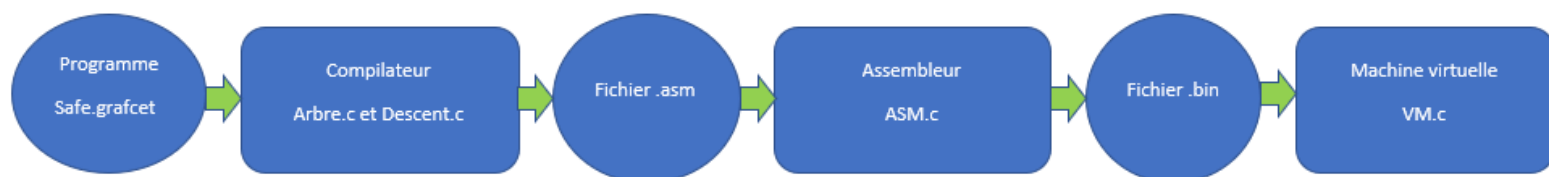
Mathieu Esteban et Stéphane Padrao  
POLYTECH SORBONNE

## Table des matières

INTRODUCTION .....	2
I) Le programme grafcet du coffre-fort en .c.....	2
II) Le programme de la machine virtuelle.....	4
1) Présentation du fonctionnement de la machine virtuelle .....	4
2) Le programme de la machine virtuelle.....	5
3) Test du programme de la machine virtuelle .....	8
III) Le programme de l'assembleur .....	9
1) Présentation du fichier .asm .....	9
2) Explications des fonctions du langage c de base utilisées dans notre programme .....	10
3) Déroulement de notre programme.....	11
4) Test du programme de l'assembleur.....	16
IV) Le programme du compilateur.....	17
1) Présentation du fonctionnement du compilateur .....	17
2) Le programme du compilateur.....	17
3) Test du programme du compilateur .....	20
CONCLUSION .....	20

## INTRODUCTION

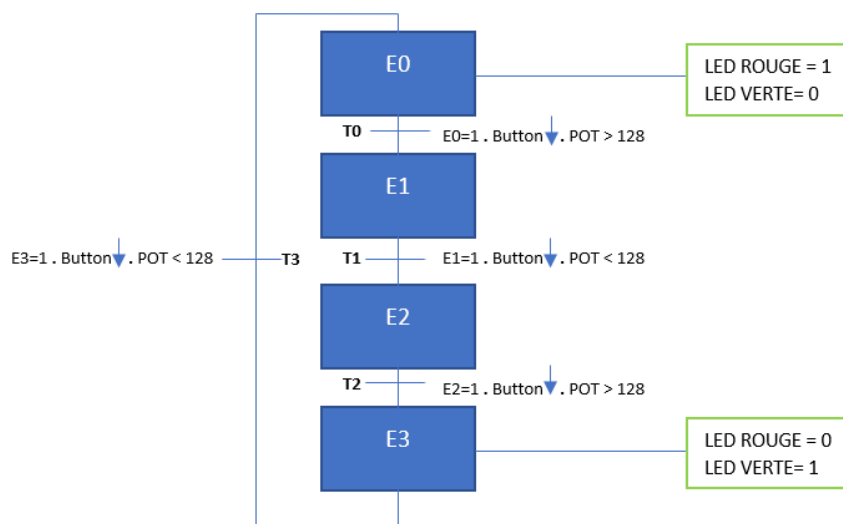
Ce projet d'informatique a consisté à programmer en C les différentes parties qui composent la compilation d'un code en grafcet. En l'occurrence ici nous nous intéressons au grafcet d'un coffre-fort qui en fonction des différentes combinaisons qui lui sont mises en entrée, passe ou non d'une étape à l'autre dans le grafcet. Ainsi, nous avons dans un premier temps programmé le grafcet de notre coffre-fort en C qui reçoit en entrée par l'intermédiaire d'un fichier .h les différentes combinaisons à tester pour déverrouiller celui-ci. Le projet a ensuite consisté à réaliser la chaîne de compilation d'un fichier grafcet que nous avons créé et qui sera mis en entrée du compilateur. Dans un premier temps nous avons donc eu à programmer une machine virtuelle, puis un assembleur et enfin un compilateur. Nous avons donc fait les étapes de traitement du fichier grafcet à l'envers. Les trois fichiers que nous avons eu à coder, une fois assemblés peuvent se représenter par le schéma ci-dessous :



*Ensemble des étapes d'interprétation du fichier « safe.grafcet »*

### l) Le programme grafcet du coffre-fort en .c

Nous avons donc réalisé un programme du grafcet en C pour le coffre-fort d'après le schéma du grafcet représenté ci-dessous :



*Grafcet du coffre-fort*

Le code en C se décompose donc ainsi :

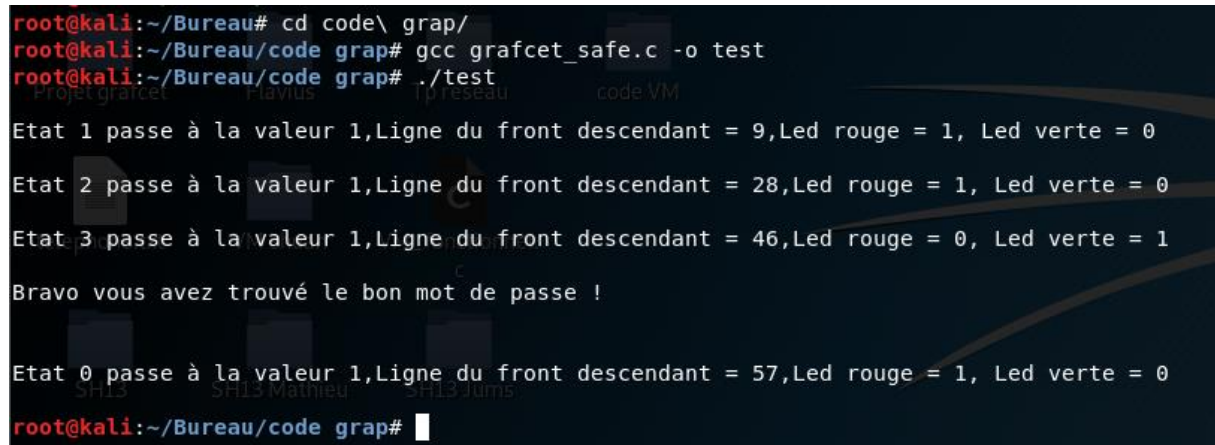
Dans un premier temps nous ouvrons le fichier « stimuli.txt » avec la fonction « fopen () », nous démarrons à l'état  $E_0$  et le compteur « cpt » à 0. Enfin, nous utilisons une boucle « while () » que nous parcourons tant que nous n'atteignons pas la valeur « (-1) » dans le fichier « stimuli.txt ». Ainsi, nous lisons chaque valeur du « stimuli.txt » avec la fonction « fscanf () ». Nous stockons chaque nouvelle valeur du bouton poussoir dans le tableau « val [2] » à l'emplacement 0 et celle du potentiomètre dans le même tableau à l'emplacement 1. La valeur du bouton poussoir passe à 0 (front descendant) lorsque le bouton poussoir est appuyé. Nous testons toutes les conditions détaillées dans le schéma du grafcet dans notre code C, qui prennent en compte la valeur de l'état, la valeur du bouton poussoir et la valeur du potentiomètre afin de changer d'état :

```
while(val[0] != (-1))           //tant qu'on arrive pas à la valeur du bouton = '-1'
{
    fscanf(fichier,"%d %d", &val[0], &val[1]); //lecture des valeurs du fichier stimuli.txt et stockage
    cpt++;                                   //on incrémente cpt qui représente le numéro de la ligne lue

    if ((ETAT==0) && (val[0]==0) && (val[1]>128))//on passe à l'état 1 si on est dans l'état 0, la valeur du bouton vaut 0
                                                //et la valeur du potentiomètre lue > 128
    {
        ETAT=1;
        printf("\nEtat 1 passe à la valeur 1,");
        printf("Ligne du front descendant = %d",cpt);
        printf("Led rouge = 1, Led verte = 0 \n");
    }
}
```

*Condition pour passer dans l'état 1*

Ainsi, après lecture du fichier « stimuli.txt » notre programme parcourt les différentes combinaisons (état bouton poussoir + valeur potentiomètre) de celui-ci pour pouvoir passer d'un état à l'autre. Le fichier « stimuli.txt » permet bien de déverrouiller le coffre-fort :



```
root@kali:~/Bureau# cd code\ grap/
root@kali:~/Bureau/code grap# gcc grafcet_safe.c -o test
root@kali:~/Bureau/code grap# ./test

Etat 1 passe à la valeur 1,Ligne du front descendant = 9,Led rouge = 1, Led verte = 0
Etat 2 passe à la valeur 1,Ligne du front descendant = 28,Led rouge = 1, Led verte = 0
Etat 3 passe à la valeur 1,Ligne du front descendant = 46,Led rouge = 0, Led verte = 1
Bravo vous avez trouvé le bon mot de passe !

Etat 0 passe à la valeur 1,Ligne du front descendant = 57,Led rouge = 1, Led verte = 0
root@kali:~/Bureau/code grap#
```

*Exécution du programme Grafcet.c*

Ce programme « grafcet.c » nous aide à comprendre ce que nous devons obtenir en sortie de notre machine virtuelle.

## II) Le programme de la machine virtuelle

### 1) Présentation du fonctionnement de la machine virtuelle

Le programme de la machine virtuelle a pour rôle d'interpréter un fichier « .bin » qui lui est mis en entrée. Ce fichier « safe.bin » contient une liste d'instruction en byte code qui traduisent le code du grafcet.c. Le fichier « .bin » donné peut se décomposer en 4 parties. Chaque partie de ce fichier correspond à un test de condition, qu'on pourrait assimiler à un « if » du fichier « grafcet.c ». Chacune de ses parties vérifie à l'aide des instructions, si l'état  $E_n$  est à 1, si un front descendant s'active pour le bouton poussoir et si la valeur du potentiomètre est inférieure ou supérieure à la valeur de référence choisie. L'objectif est donc d'interpréter chaque instruction du fichier byte code qui se présente comme ceci :

```

109
0:100 //stocke 1 dans la pile
1:1
2:200 // lire et stocker dans la pile la valeur de l'ETAT
3:0
4:0
5:407 //on regarde si on est bien dans l'état 1 par une égalité et on stocke 1 si oui et 0 si non dans la pile
6:200 //lire et stocker dans la pile la valeur de l'état du front (montant ou descendant) du bouton
7:4
8:2
9:404 //on fait un et logique entre la valeur du bouton et la valeur de sp précédente (résultat à 1 si on a un front descendant et E0=1)
10:100 //lire et stocker dans la pile la valeur du pot égal à 128 (valeur de ref)
11:128
12:200 //lire et stocker dans la pile la valeur du pot du tableau
13:5
14:0
15:408 // test si valeur pot sup à 128 et 1 si oui et 0 si non (résultat stocké dans la pile)
16:404 //on fait un et logique entre la valeur 1 ou 0 résultant de la val du potentiomètre et la valeur de sp précédente
17:500 // condition if
18:27
19:100 //stocke 0 dans la pile
20:0
21:300 // stocke le 0 dans le tableau à l'etat E0
22:0
23:100 //stocke 1 dans la pile
24:1
25:300 // stocke le 1 dans le tableau à l'etat E1
26:1
  
```

*Extrait du fichier « safe.bin » commenté permettant de passer de l'état 0 à l'état 1*

Ainsi, la première ligne de ce fichier correspond au nombre d'instructions présentes dans le fichier. Nous avons ici 109 instructions dans le fichier « safe.bin », à chaque ligne il y a deux nombres séparés par « : ». Le premier nombre à gauche correspond à l'index et celui à droite des « : » correspond au code d'opération qui est propre à chaque instruction. Ces différentes opérations correspondent à des fonctions usuelles du type addition, soustraction, insertion de données depuis un tableau ou une entrée physique, des comparateurs et des sauts dans la pile. Par conséquent, en utilisant le fichier appelé « vm\_codops.h », nous pouvons faire le lien entre le code et l'instruction souhaitée :

```

#define OP_HALT 1000 //marquage de fin du fichier .bin
#define OP_PUSHI 100//insertion d'une valeur d'un élément extérieur dans la pile
#define OP_PUSH 200//insertion d'une valeur dans la pile
#define OP_POP 300//envoi d'une valeur de la pile vers l'extérieur
#define OP_PLUS 400//addition de deux valeurs de la pile
#define OP_MOINS 401//soustraction de deux valeurs de la pile
#define OP_MULT 402//multiplication de deux valeurs de la pile
#define OP_DIV 403//division de deux valeurs de la pile
#define OP_AND 404//opération d'un ET logique entre deux valeurs de la pile
#define OP_OR 405//opération d'un OU logique entre deux valeurs de la pile
#define OP_NOT 406//opération d'un NOT logique sur une valeur de la pile
#define OP_EQ 407//vérification de l'égalité entre deux valeurs de la pile
#define OP_GT 408//vérifie qu'une valeur de la pile est plus grande qu'une autre valeur
#define OP_LT 409//vérifie qu'une valeur de la pile est plus petite qu'une autre valeur
#define OP_JZ 500//fait un saut dans la pile si la valeur de la pile actuelle vaut 0
#define OP_JP 501//fait un saut dans la pile si la valeur de la pile actuelle vaut 1
  
```

*Nom des instructions en lien avec les codes du fichiers « safe.bin »*

On peut alors observer que 1000 correspond à l’instruction « OP\_HALT », que 100 correspond à l’instruction « OP\_PUSHI », que 300 correspond à l’instruction « OP\_POP » etc...

Nom	Valeur	↑	↓
E0	1	-	-
E1	0	-	-
E2	0	-	-
E3	0	-	-
BUTTON	-	X	X
POT	X	-	-
LED ROUGE	1	-	-
LED VERTE	0	-	-

Tableau des éléments du grafcet

X : valeur inconnue qui change au cours de la lecture du fichier « stimuli.txt ».

## 2) Le programme de la machine virtuelle

Dans un premier temps notre programme va ouvrir dans la fonction « main () » le fichier « safe.bin » en mode lecture. Ensuite une vérification a lieu pour s’assurer que le fichier ouvert existe bien, si le programme n’arrive pas à l’ouvrir alors un message d’erreur s’affiche :

```
FILE* fin=fopen("safe.bin","r");//ouvre "safe.bin" en r
if (fin==NULL)//si le fichier "safe.bin" n'existe pas,
{
    printf("Erreur ouverture fichier %s\n",argv[1]);
    exit(1);
}
```

Ouverture du fichier « safe.bin » en mode lecture

Ensuite, il faut stocker l’ensemble des valeurs d’instruction du fichier « safe.bin » dans un tableau afin de pouvoir les lire et les interpréter ensuite. Ainsi, nous avons initialisé un tableau « code [] » et lu le fichier à l’intérieur d’une boucle « for » afin de stocker les valeurs d’instruction :

```
fscanf(fin,"%d", &NombreLigne);//on lit la première ligne
/*tableau pile qui stockera les valeurs dans la pile et t
int pile[NombreLigne], code[NombreLigne];

for (int i = 0; i < NombreLigne; ++i)//on remplit avec ce
{
    fscanf(fin,"%d:%d", &ligne, &code[i]);
}
```

Stockage des instructions du fichier « safe.bin »

Après avoir stocké les valeurs des instructions, nous ouvrons en lecture le fichier « stimuli.txt ». Puis nous appelons la fonction « run\_VM () » à laquelle nous mettons en argument notre fichier en lecture « stimuli.txt », notre tableau « pile [] » qui va stocker les valeurs de la pile issue des instructions du fichier « .bin » et notre tableau « code [] » déjà rempli :

```
FILE* fichier=fopen("stimuli.txt","r");//ou  
run_VM(fichier, pile, code);
```

*Appel de la fonction « run\_VM » qui va interpréter les instructions*

Une fois dans la fonction « run\_VM () » nous initialisons un tableau « MTR [8][3] » représenté plus haut, qui contiendra les éléments du grafcet tels que l'état des états E0, E1, E2 et E3, l'état du bouton poussoir, des leds et la valeur du potentiomètre. Nous initialisons les valeurs des éléments du tableau comme ci-dessous :

```
void run_VM(FILE *fichier, int pile[], int code[])//fonction qui va lire chaque ligne du stimu  
//les valeurs du BP et du p  
{  
    int MTR[8][3]; //tableau contenant les valeur des états, du bouton poussoir  
    //du potentiomètre et des LEDs  
  
    MTR[0][0]=1; //on démarre avec l'état 0 (E0) à la valeur 1  
    MTR[1][0]=0; //état 1 (E1) à la valeur 0  
    MTR[2][0]=0; //état 2 (E2) à la valeur 0  
    MTR[3][0]=0; //état 3 (E3) à la valeur 0  
    MTR[6][0]=1; //LED rouge à la valeur 1 (allumée)  
    MTR[7][0]=0; //LED verte à la valeur 0 (éteinte)  
    int *L_R= &(MTR[6][0]); //on déclare L_R (LED rouge) qui pointe à l'adresse de "MTR[6][0]"  
    int *L_V= &(MTR[7][0]); //on déclare L_V(LED verte) qui pointe à l'adresse de "MTR[7][0]"
```

*Extrait de code d'initialisation du tableau du grafcet*

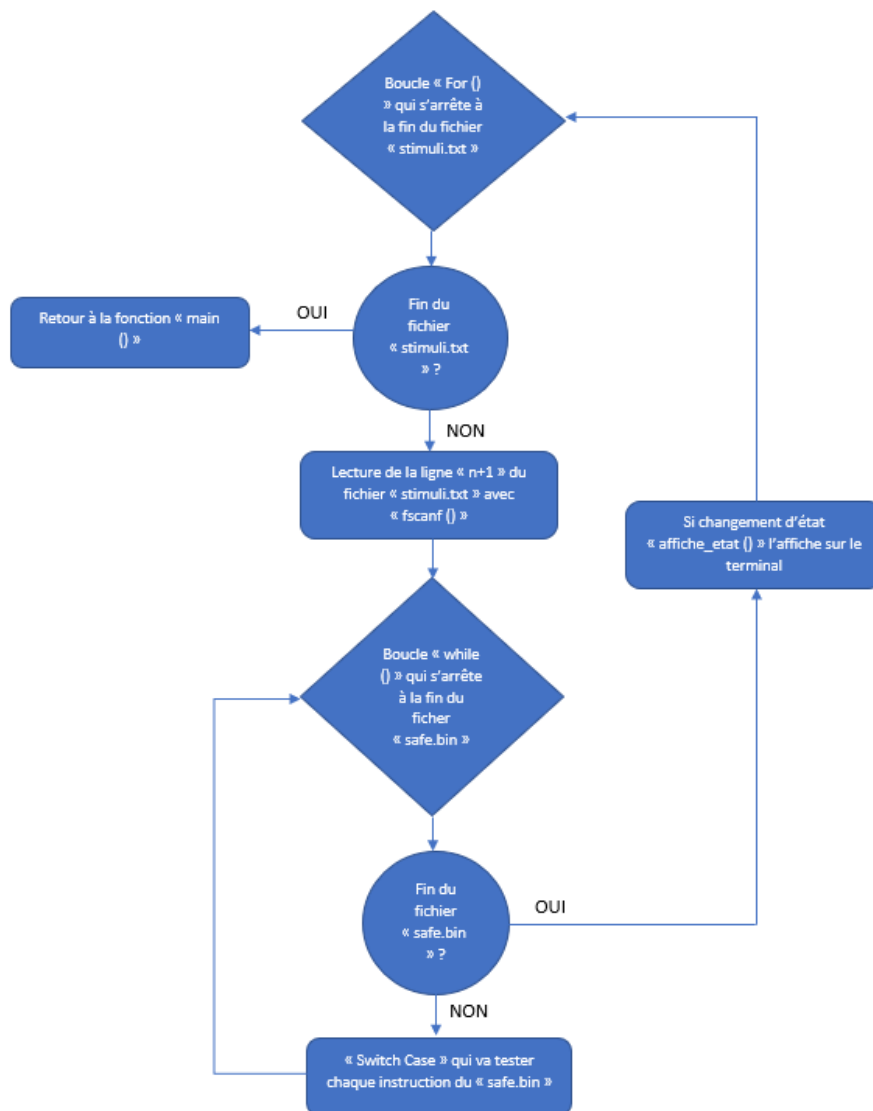
Ensuite, nous initialisons un tableau « val [2] » dans lequel seront stockées les valeurs du bouton poussoir et du potentiomètre issues de la lecture du fichier « stimuli.txt ». Nous initialisons également une variable de type entier « old\_button » à 0 qui servira à détecter un front montant. Les variables suivantes initialisés sont :

- a et b : variables de type entier qui vont servir de tampon lorsqu'on voudra effectuer une opération entre deux valeurs de la pile.
- pc : variable de type entier qui servira d'index du tableau « code [] » pour savoir quelle instruction sera souhaité par le fichier « safe.bin ». Celle-ci sera incrémenté après chaque passage d'une instruction à une autre.
- sp : variable de type entier qui gère l'index de la pile.

```
int val[2]; //tableau qui va contenir les valeur de BP et du potentiomètre  
int old_button=0; //on initialise la valeur de l'appui précédent à 0  
  
int a,b,pc,sp;
```

*Déclaration de différentes variables*

Après cela nous devons faire tester chaque ligne du « stimuli.txt » par notre « safe.bin ». En effet le « safe.bin » est notre code du programme grafcet du coffre-fort écrit en bytecode. Celui-ci doit donc tester les valeurs du bouton poussoir et du potentiomètre pour passer de l'état «  $E_n$  » à l'état «  $E_{n+1}$  ». La suite du code de la fonction « run\_VM () » se décompose donc ainsi :



Ordinogramme de la fonction « run\_VM () »



Ainsi, chaque instruction du fichier « safe.bin » est testé et si les conditions de changement d'état ( $E_n$  à 1, bon état du bouton poussoir et bonne valeur du potentiomètre) sont bonnes alors la dernière valeur de « pile[sp] » doit valoir 1 lorsqu'on arrive à l'instruction « JZ ». Cette instruction renvoie ensuite aux instructions suivantes qui permettent de réaliser le changement d'état. Nous affichons ensuite avec la fonction « affiche\_etat () » ce changement d'état :

```
// saut JZ
case OP_JZ :

    if(pile[sp]==1) //si toutes les conditions pour changer d'état sont respectées pile[sp]=1
    {
        pc=pc+2; //on va vers l'instruction après JZ (ces instructions entraînent le changement d'état)
    }
    else
    {
        pc++;
        pc = code[pc]; // on saute directement à l'autre bloc d'instruction qui va tester si on est dans un l'état En
        //on saute ici 8 instructions d'un coup
    }
break;
```

*Instruction JZ de notre programme*

Afin d'optimiser la taille des piles, la valeur des variables « pc » et « sp » sont remises à 0 après chaque fin de lecture du fichier « safe.bin ».

### 3) Test du programme de la machine virtuelle

Le programme de la machine virtuelle « VM.c » a donc pu être testé sous linux et nous obtenons bien le même résultat que notre programme de référence « grafcet.c », ce qui confirme que le code de la machine virtuelle fonctionne correctement :

```
root@kali:~/Bureau/code VM# gcc VM_avec_fonctions_05_06_19.c -o fin
root@kali:~/Bureau/code VM# ./fin
Etat 1 passe à la valeur 1
Ligne du front descendant = 9,
Led rouge = 1, Led verte = 0

Etat 2 passe à la valeur 1
Ligne du front descendant = 28,
Led rouge = 1, Led verte = 0

Etat 3 passe à la valeur 1
Ligne du front descendant = 46,
Led rouge = 0, Led verte = 1

Félicitation vous avez trouvé le bon mot de passe !

Etat 0 passe à la valeur 1
Ligne du front descendant = 57,
Led rouge = 1, Led verte = 0
```

*Exécution du programme de la VM*

Les instructions du fichier bytecode ont donc été correctement interprétés.

### III) Le programme de l'assembleur

Un langage assembleur (asm) est un langage de programmation de très bas niveau dans lequel il existe une correspondance entre les instructions du programme et les instructions du code machine. Chaque langage d'assemblage est spécifique à une architecture d'ordinateur particulière ou à un système d'exploitation. La plupart des langages d'assemblage peuvent être utilisés universellement avec n'importe quel système d'exploitation, car le langage fournit un accès à toutes les fonctionnalités réelles du processeur, sur lequel reposent tous les mécanismes d'appel système.

Un assembleur est un programme qui interprète le code de programmation écrit en langage assembleur en code machine (dans notre cas en .bin) et en instructions qui peuvent être exécutés par la machine.

Dans cette partie nous allons étudier le fonctionnement de notre assembleur spécifique à notre système. Nous devons alors convertir un fichier « .asm » en un fichier « .bin » qui sera par la suite lui-même interprété.

#### 1) Présentation du fichier .asm

Pour notre projet, nous avons un fichier de test « safe.asm » qui correspond au code assembleur que nous devons interpréter. Présentons comment le fichier est formé :

```

1      pushi 1
2      push 0 0
3      eq
4      push 4 2
5      and
6      pushi 128
7      push 5 0
8      gt
9      and
10     jz suite0
11     pushi 0
12     pop 0
13     pushi 1
14     pop 1
15 suite0:
```

*Extrait du code à interpréter*

Voici un extrait du fichier d'exemple, avec quelques explications des « fonctions » utilisé dedans.

Pushi = fonction qui met la valeur 1 dans la pile.

Push = fonction qui met une valeur (0 ici) à la position 0.

And = fonction qui addition les 2 dernières valeurs de la pile.

JZ = fonction qui correspond à un saut PC au niveau de la pile.

Nous allons alors à présent présenter les fonctions de base de la manipulation de fichiers du langage c.

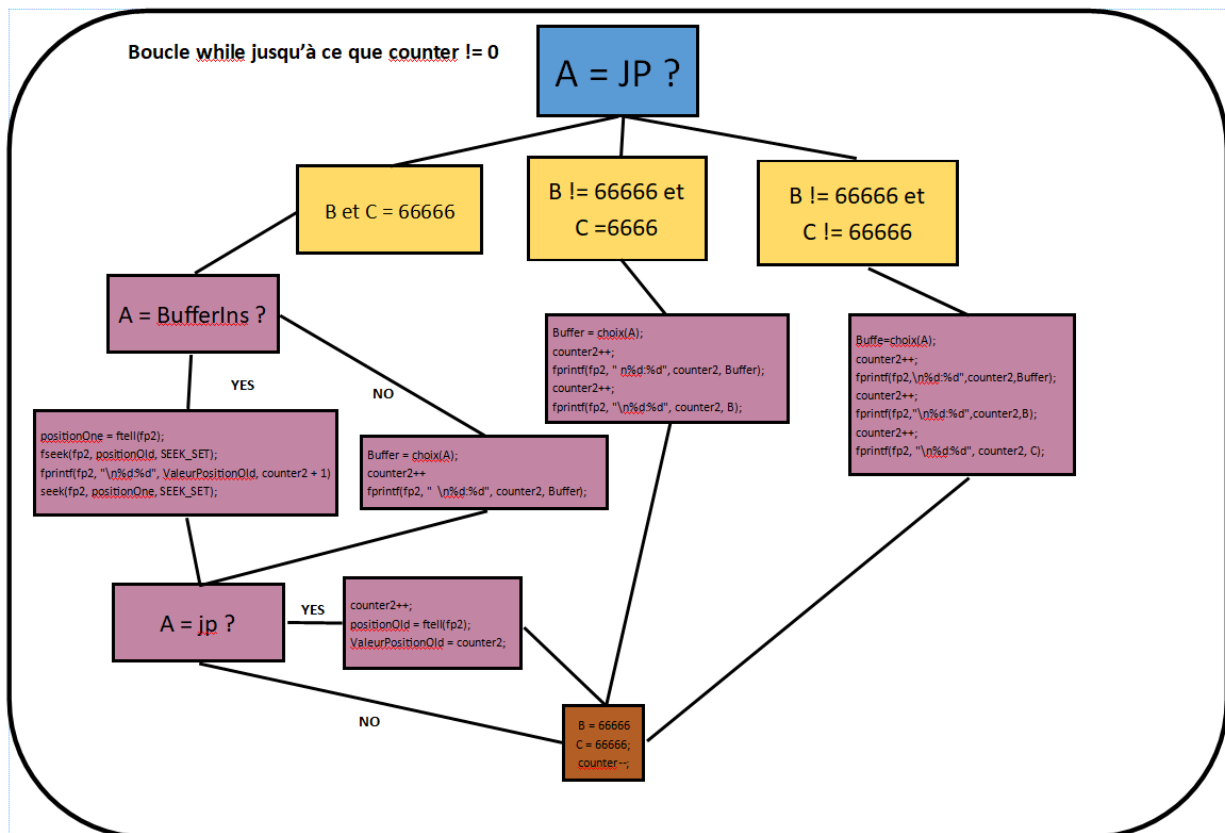
## 2) Explications des fonctions du langage c de base utilisées dans notre programme

- FILE\* fp : fp est un pointeur sur une structure type FILE. La gestion des fichiers se trouve dans la librairie stdio.h
- fopen(argv[1], "r") :
  - FOPEN récupère la valeur du fichier renvoyé dans le pointeur FILE.
  - Le paramètre argv[1] est un tableau de pointeurs, qui pointe sur des chaînes de caractère. Cela nous permet d'indiquer dans le terminal au lancement du programme le nom du code que nous souhaitons interpréter.
  - « r », est une option de la fonction FOPEN. Dans notre cas elle n'autorise uniquement la lecture du fichier. D'autre option existe comme « w » qui est permet d'uniquement écrire dans le fichier.
- Strlen : fonction qui nous permet de connaître la longueur d'une chaîne de caractère.
- Fgets : nous permet de lire une ligne de caractères du fichier.
- Strcat : fonction qui nous permet de rajouter à une chaîne de caractères déjà existante le contenu d'une seconde chaîne.
- Rewind : fonction qui nous permet de nous positionner au début d'un fichier.
- Fscanf : fonction qui permet la lecture de texte suivant un certain format dans un fichier.
- Ftell : fonction qui nous retourne la position du pointeur dans le fichier texte.
- Fseek : fonction qui nous permet de nous positionner à un endroit précis du fichier.
- Fprintf : fonction qui effectue l'écriture de texte selon un certain format dans un fichier.

Voici la liste des fonctions les plus utilisées au sein du programme, nous allons à présent présenter le fonctionnement de notre programme.

### 3) Déroulement de notre programme

Voici la boucle principale du fonctionnement de l'assembleur :



Le début de notre programme est composé d'une série d'initialisation de variable, tableau, de pointeurs fichiers, de manière à préparer notre boucle d'interprétation principale.

```

short counter = 0;
char line[50];
char* filename;

char A[10];
int B = 66666;
int C = 66666;

FILE* fp = fopen(argv[1], "r"); //ouverture
if(fp == NULL)
{
    printf("\n Impossible d'ouvrir le fichier \n");
    exit(1);
}
  
```

*Initialisation de variables*

On déclare les variables utiles au programme. Les variable B et C sont des variable buffer dont leur but est de contenir les différentes variables décimales contenue dans le code à interpréter. Elles sont initialisées à 66666 qui est une valeur aléatoire, qui on le sait n'est jamais utilisée dans notre système.

A est une variable « char » de buffer du nom de l'instruction à interpréter du fichier « .asm ».

On ouvre le fichier à interpréter grâce à « fopen () » et on a créé un pointeur pointant dessus avec « FILE \* fp ». On pourra grâce à « argv[1] » obtenir le nom du fichier lorsque celui-ci sera tapé dans le terminal.

On teste si le fichier est vide ou non grâce au « IF » ; Si le fichier est vide il doit être égale à NULL et donc on affiche un message d'erreur.

```
filename = argv[1]; //chaîne qui prend le nom du fichier entre e
for (size_t i = strlen(argv[1]) - 4; i < strlen(argv[1]); i++)
{
    printf("%d\n", filename[i]);
    if (filename[i] == '.') {}
    else if (filename[i] == 'a') {}
    else if (filename[i] == 's') {}
    else if (filename[i] == 'm') {}
    else
    {
        printf("erreur fichier input check le type de fichier\n");
        exit(1);
    }
}
filename[strlen(argv[1]) - 4] = 0; //efface caractère a par
```

*Vérification du type de fichier en entrée*

Notre variable précédemment créée appelée « filename » prend le nom du fichier du terminal.

On entre dans une boucle for qui nous permet en décomposant la chaîne de caractère de déterminer si le fichier se termine bien par « .asm ». Si cela n'est pas le cas on stop le programme et on affiche une erreur.

Notre interpréteur offre les avantages de pouvoir entrer le programme à interpréter directement depuis le terminal (à condition que celui-ci soit dans le même dossier que le programme lui-même) et de vérifier qu'il s'agit bien d'un fichier « .asm ».

Avec la fonction « strlen () », on efface les quatre derniers caractères de la chaîne du nom du fichier pour pouvoir utiliser le nom du fichier pour le fichier de sortie. Cela nous permet de nous débarrasser de l'extension « .asm ».

```
while(fgets(line, sizeof(line), fp) != NULL)
{
    counter++;
}

counter -= 1;

strcat(filename, ".bin"); //Mo
FILE* fp2 = fopen(filename, "w+");

rewind (fp);
//positionOne = ftell(fp2);
Détermination du nombre de ligne du fichier d'entrée et
préparation du fichier de sortie
```

La boucle « while () » nous permet de compter le nombre de lignes (d'instructions) que le code contient. Pour cela on utilise la fonction « fgets () » qui va lire chaque chaîne de caractère du code et passer à ligne suivante quand une ligne est terminée. Ainsi, lorsqu'on sera à la dernière ligne vide on le saura car le fichier renverra une valeur NULL. On sortira alors de la boucle et on gardera en mémoire le nombre de ligne.

La fonction « strcat () » nous permet de rajouter à la chaîne de caractère du nom du fichier l'extension « .bin ».

```

while(counter != 0)                                //
{
    fscanf(fp, "%s %d %d", A, &B, &C);            //

    if ((strcmp(A, "jp") == 0) || (strcmp(A, "jz") == 0))
    {
        fscanf(fp, "%s", BufferIns);              //
        strcat(BufferIns, ".");                  //
    }
}

```

*Test si la variable A est un jump*

On rentre dans la boucle d'interprétation du code. On utilisera la variable Counter précédemment initialisée avec le nombre de lignes du code pour déterminer quand est-ce qu'on aura fini d'analyser tout le fichier. Lorsque Counter sera à 0 on sortira de la boucle.

La première étape de l'interprétation est de récupérer toutes les valeurs, chaînes de caractères contenus sur une ligne du code. Pour cela on utilise la fonction « `fscanf()` » qui stockera chaque chaîne, valeurs dans une variable différentes (A, B, C qui sont précédemment détaillées).

Premier test avec la condition « IF », on cherche à déterminer si la variable A est égale à JP ou JZ. Pour rappel les instructions JP ou JZ correspondent à des « jump » d'instruction dans la mémoire. Pour cela on cherche à déterminer en premier si notre ligne du code correspond à un de ces cas.

Si nous sommes bien dans la situation d'un « jump », on enregistre la chaîne de caractère qui correspond au jump. Par exemple on peut avoir « JZ suite1 » on retient alors « suite1 » pour plus tard retrouver où se situe le « jump » dans le code.

```

if((B == 66666) && (C == 66666))                  //comparaison
{
    if (strcmp(A, BufferIns) == 0)                 //test si A es
    {
        positionOne = ftell(fp2);                 //recupere la
        fseek(fp2, positionOld, SEEK_SET);         //modifie la p
        fprintf(fp2, "\n%d:%d", ValeurPositionOld, counter2 + 1);
        fseek(fp2, positionOne, SEEK_SET);         //reposition l
    }
    else                                           //si la compar
    {
        Buffer = choix(A);                         //choix de la
        counter2++;
        fprintf(fp2, "          \n%d:%d", counter2, Buffer); //
    }

    if ((strcmp(A, "jp") == 0))                   //test de jp et jz pou
    {
        counter2++;
        positionOld = ftell(fp2);                 //position
        ValeurPositionOld = counter2;              //valeur d
    }
}

```

*Premier test pour l'interprétation du code*

La deuxième étape correspond à une lecture de ligne classique pour les autres cas d'instruction qui ne sont pas particulières. On teste si les variables enregistrées sont égales à 66666. Si cela est le cas cela signifie que l'on se trouve dans une instruction de type JP, GT ou encore LT.

On teste alors si la variable est égale à la variable « BufferIns ». Cette variable nous a servi à enregistrer le nom du « jump ». Si on est dans ce cas on réalise :

- On enregistre la position dans le texte où l'on se trouve.
- On se positionne dans le deuxième fichier (.bin) à la position où le jump a été enregistré.
- On écrit dans le fichier de sortie l'instruction interprétée, avec son indexation.
- On se positionne à la position originale.

Sinon, si le « IF » n'est pas réalisé on réalise :

- Enregistre dans une variable « Buffer » l'instructions interprété.  
On interprète l'instruction lue dans le code en utilisant une fonction. On détaillera cette fonction plus loin dans le rapport.
- On incrémente le compteur2 qui gère l'indexation des instructions interprétées pour le fichier de sortie.
- On écrit dans le fichier de sortie la valeur index plus l'instructions interprétée.

Exemple : Si l'on a « pushi 1 » on obtient « 1:1 » pour la valeur et « 0:100 » pour l'instruction

Toujours dans la deuxième étape on réalise un autre test pour savoir si la variable A est égale à JP.

Ainsi, on réalise :

- On incrémente le compteur 2.
- La variable de « positionOld » prend la position actuelle dans le fichier.
- On enregistre dans une variable Buffer « ValeurPositionOld » la valeur du compteur 2.

Si le premier test « IF » n'est pas concluant et donc qu'au moins une variable à changer. On réalise le test « IF » suivant :

```
else if((B != 66666) && (C == 66666))           //
{
    Buffer = choix(A);                             //
    counter2++;
    fprintf(fp2, "          \n%d:%d", counter2, Buffer);
    counter2++;
    fprintf(fp2, "\n%d:%d", counter2, B);           //
}
```

*Deuxième test pour l'interprétation du code*

Le « IF » teste si la variable B a été modifiée mais pas la variable C !

Si cela est le cas on réalise donc :

- On interprète l'instruction du code en utilisant la fonction dédiée.
- Compteur 2 s'incrémente.
- On écrit dans le fichier de sortie, la variable interprétée et son indexation à l'aide du compteur.
- On incrémente le compteur pour les futures écritures de fichier.
- On écrit dans le fichier de sortie la variable liée à l'instruction interprétée.

Si le deuxième test « IF » n'est pas concluant et donc qu'au moins une variable à changer. On réalise le test « IF » suivant :

```
else if((B != 66666) && (C != 66666))
{
    Buffer = choix(A);
    counter2++;
    fprintf(fp2, "          \n%d:%d", counter2, Buffer);
    counter2++;
    fprintf(fp2, "\n%d:%d", counter2, B);
    counter2++;
    fprintf(fp2, "\n%d:%d", counter2, C);
}
```

*Troisième test pour l'interprétation du code*

Le test « IF » teste si les variables B et C sont modifiées. Ainsi on aura vérifié tous les cas possibles.

On réalise alors :

- L'interprétation de l'instruction du code.
- On incrémente le compteur 2 une fois.
- On écrit dans le fichier de sortie l'interprétation de l'instructions du code avec son indexation.
- On incrémente le compteur encore une fois.
- On écrit dans le fichier de sortie la première valeur liée à l'instruction et son indexation.
- On incrémente le compteur une dernière fois.
- On écrit dans le fichier de sortie la deuxième valeur liée à l'instruction et son indexation.

Après la réalisation de ses différents tests, nous avons testé tous les cas possibles pour l'interprétation de notre code

```

        B = 66666;
        C = 66666;
    counter--;
}
    
```

*Réinitialisations des variables buffer*

On remet les valeurs buffer de tests aux valeur initiales et on soustrait au compteur principal – 1 ; de cette façon au prochain tour de notre boucle on testera la ligne suivante.

Fonction d'interprétation du code :

Voici un extrait de la fonction, on ne montre que quelques interprétations, le reste étant identique :

```

short choix(char A[50])    //fonction
{
    short Retour;
    if(strcmp(A, "pushi") == 0)
    {
        Retour = OP_PUSHI;
    }
    else if (strcmp(A, "push") == 0)
    {
        Retour = OP_PUSH;
    }
    else if (strcmp(A, "pop") == 0)
    {
        Retour = OP_POP;
    }
}
    
```

*Extrait de la fonction interprétation du code*

On récupère la chaîne de caractère A.

On teste par rapport à de nombreuses autres instructions interprétées.

On renvoie la valeur de cette interprétation sous la forme d'une variable « short ».

Une librairie contenant toutes les valeurs interprétées nous a été fournie. On l'utilise en l'appelant à l'aide « #include "vm\_codops.h" ».



#### 4) Test du programme de l'assembleur

On compile le programme et la librairie de la façon suivante :

```
File Edit View Search Terminal Help
sp@sp-VirtualBox:~/Desktop/PROJET INFO 6 ASSEMBLEUR$ gcc Asm.c vm_codops.h -o main
```

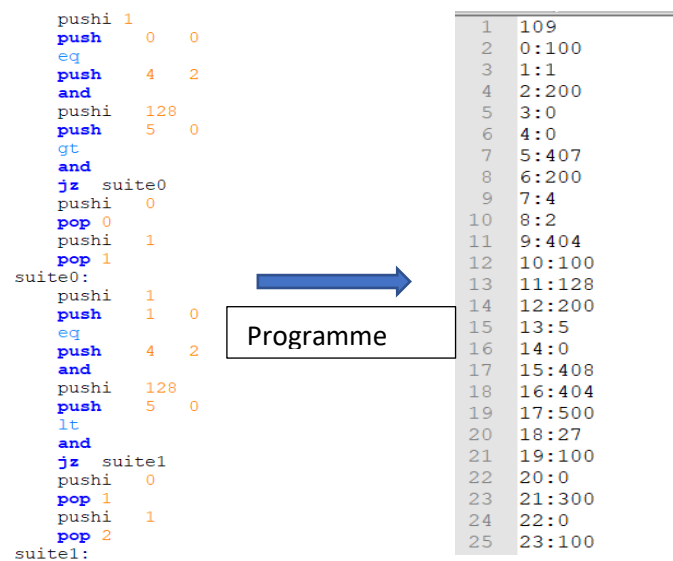
*Compilation du programme*

Puis on lance le programme en appelant en argument le nom du fichier à interpréter :

```
File Edit View Search Terminal Help
sp@sp-VirtualBox:~/Desktop/PROJET INFO 6 ASSEMBLEUR$ ./main safe.asm
sp@sp-VirtualBox:~/Desktop/PROJET INFO 6 ASSEMBLEUR$ ls
Asm.c  main  safe.asm  safe.bin  vm_codops.h
sp@sp-VirtualBox:~/Desktop/PROJET INFO 6 ASSEMBLEUR$
```

*Lancement du programme assembleur et création du fichier « safe.bin »*

On remarque que le fichier « safe.bin » a bien été généré :



On peut conclure que notre assembleur fonctionne comme attendu. Celui-ci a été testé avec le programme de la machine virtuelle et permet d'obtenir la bonne exécution du grafctet.

## IV) Le programme du compilateur

### 1) Présentation du fonctionnement du compilateur

Par manque de temps nous n'avons pas pu finaliser cette partie. Cependant, nous expliquerons ce que nous avons compris de l'objectif du compilateur, la méthode pour le réaliser et une partie du programme « arbre.c » complété.

Pour réaliser la partie compilateur de la chaine, nous avons créé un fichier texte appelé « safe.grafcet » contenant les étapes précise pour passer d'un état à un autre :

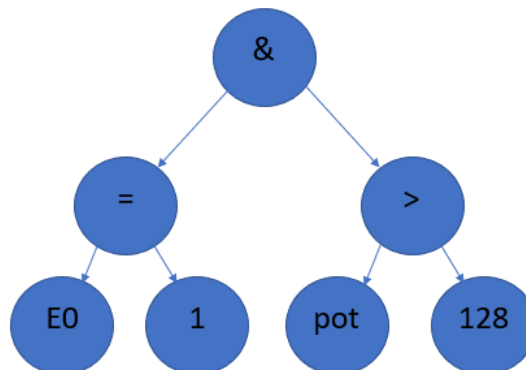
```
E0=1 & bp=0 & 128 > pot
E1=1 & bp=0 & 128 < pot
E2=1 & bp=0 & 128 > pot
E3=1 & bp=0 & 128 < pot
end
```

L'objectif du compilateur est qu'à partir du fichier « safe.grafcet » créé, il faut passer d'une écriture infixe à une écriture postfixe. L'écriture postfixe est l'écriture dans laquelle est écrit l'assembleur et aussi celle dont a besoin la machine virtuelle pour exécuter ses instructions. Ainsi, l'écriture de ce fichier en écriture postfixe serait :

Ligne 1 :	Ligne 2 :	Ligne 3 :	Ligne 4 :
E0 1 =	E1 1 =	E2 1 =	E3 1 =
and	and	and	and
bp 0 =	bp 0 =	bp 0 =	bp 0 =
and	and	and	and
pot 128 >	pot 128 <	pot 128 >	pot 128 <

### 2) Le programme du compilateur

Le programme « arbre.c » qui nous a été fourni, doit pouvoir générer un Abstract Syntax Tree (AST) en sortie. Cet AST est une représentation du programme grafcet mis en entrée sous forme d'une arborescence. Ainsi la représentation en AST de « E0=1 & pot>128 » sera :



Ensuite cet arbre généré par création de nœuds devra être parcouru par une fonction « `parcours()` » qui créera à partir de celui-ci le code assembleur.

Code de création de nœuds dans le « `main()` », pour la ligne « `E0=1 & bp=0 & pot > 128` » :

```
int main(int argc, char **argv)
{
    // Code déjà présent, non utile
    noeud *pt3=creerNoeud(VARIABLE, 3, NULL, NULL);
    noeud *pt4=creerNoeud(VARIABLE, 4, NULL, NULL);
    noeud *ptAdd=creerNoeud(OPERATEUR_ADD, -1, pt3, pt4);
    noeud *pt5=creerNoeud(VARIABLE, 5, NULL, NULL);
    noeud *ptMult=creerNoeud(OPERATEUR_MULT, -1, ptAdd, pt5);
    //parcours(ptMult);

    /// Ajout de code création de noeud et appel de parcours()

    //E0 = 1
    noeud *val_1 = creerNoeud(VARIABLE, 1, NULL, NULL);
    noeud *E0 = creerNoeud(ETAT, 0, NULL, NULL);
    noeud *egalitel = creerNoeud(OPERATEUR_EQ, 1, val_1, E0);

    //& bp = 0
    noeud *bp = creerNoeud(BP, 4, NULL, NULL);
    noeud *et_1 = creerNoeud(OPERATEUR_AND, -1, egalitel, bp );

    //& pot > 128
    noeud *val_128 = creerNoeud(VARIABLE, 128, NULL, NULL);
    noeud *pot = creerNoeud(POT, 5, NULL, NULL);
    noeud *egalite2 = creerNoeud(OPERATEUR_GT, 1, val_128, pot);

    noeud *et_2 = creerNoeud(OPERATEUR_AND, -1, et_1, egalite2);

    parcours(et_2);

    return 0;
}
```

*Création de nœuds dans la fonction « `main()` »*

Nous mettons en argument de la fonction « `parcours()` » le nœud « `et_2` » correspondant à la dernière instruction « `AND` » de la première partie du fichier « `.asm` » servant à vérifier si on peut passer de l'état  $E_0$  à l'état  $E_1$ . Ainsi, l'arborescence de l'arbre va être parcouru de bas en haut grâce à la récursivité de la fonction « `parcours()` ».

Code de la fonction « `parcours ()` », qui parcourt les nœuds créés pour écrire les instructions en langage assembleur :

```
void parcours(noeud *pt)
{
    if (pt->montype==VARIABLE)
    {
        printf("\tPUSHI %d\n",pt->valeur); //on affichera l'instruction PUSHI suivie de la valeur
    }

    else if (pt->montype==ETAT)
    {
        printf("\tPUSH %d 0\n",pt->valeur); //on affichera l'instruction PUSH n 0
        //n étant l'état actuel (En)
    }

    else if (pt->montype==OPERATEUR_EQ)
    {
        parcours(pt->fg); //on regarde son fils gauche
        parcours(pt->fd); //on regarde son fils droit
        printf("\tEQ\n"); //on affichera l'instruction EQ
    }

    else if (pt->montype==POT)
    {
        printf("\tPUSH %d 0\n",pt->valeur); //on affichera l'instruction PUSH 5 0
    }

    else if (pt->montype==BP) //si le type est le bouton poussoir
    {
        printf("\tPUSH %d 2\n",pt->valeur); //on affichera l'instruction PUSH 4 2
    }

    else if (pt->montype==OPERATEUR_GT)
    {
        parcours(pt->fg);
        parcours(pt->fd);
        printf("\tGT\n"); //on affichera l'instruction GT
    }

    else if (pt->montype==OPERATEUR_AND)
    {
        parcours(pt->fg); //on regarde fils gauche
        parcours(pt->fd); //on regarde fils droit
        printf("\tAND\n"); //on affichera l'instruction AND
    }
}
```

*Fonction « `parcours ()` » qui parcourt l'arborescence de l'arbre*

Nous remarquons que cette fonction utilise beaucoup la récursivité afin de parcourir l'arborescence de l'arbre. Cette récursivité est utilisée pour parcourir les nœuds de fils gauche et fils droit. Dans ce code les instructions sont seulement affichées sur le terminal.

La partie du fichier assembleur « .asm » en sortie qui est attendue par ce code rajouté est :

```
pushi 1
push 0 0
eq
push 4 2
and
pushi 128
push 5 0
gt
and
```

### 3) Test du programme du compilateur

Nous avons donc compilé et testé notre début de code de « arbre.c » sous Linux :



```
root@kali:~/Bureau/code compilateur# gcc arbre.c -o test
root@kali:~/Bureau/code compilateur# ./test
PUSHI 1
PUSH 0 0
EQ
PUSH 4 2
AND
PUSHI 128
PUSH 5 0
GT
AND
```

Les instructions observées sur le terminal sont bien celles attendues. Les instructions sont affichées dans le bon ordre, ce qui montre que le parcours de l'arbre, par la fonction très récursive « parcours () » fonctionne correctement. Nous aurions certainement pu terminer le programme du compilateur mais nous manquons de temps malheureusement.

## CONCLUSION

Ce projet aura vraiment été très intéressant, nous avons pu en apprendre plus sur la chaîne qu'un programme doit traverser avant d'être interprété au plus bas niveau par la machine. La machine virtuelle a été le programme le plus long à réaliser. En effet, la gestion de la pile était vraiment à bien maîtriser. Il a donc fallu un long temps de débogage et de compréhension du « safe.bin » avant d'arriver à un programme fonctionnel. Nous n'avons malheureusement pas eu le temps de terminer le compilateur, mais les principes les plus importants ont nous pensons été compris. Ce projet nous a permis de progresser encore un peu plus en C.