TD 10-11 – Analyse sémantique et génération de code

1 Introduction

Dans ce TD, en s'appuyant sur l'analyse lexicale et l'analyse syntaxique déjà réalisées, nous allons programmer l'analyse sémantique et la génération de code.
En général, dans un compilateur, l'analyse sémantique et la génération de code sont deux étapes séparées, ce qui permet d'obtenir un compilateur plus modulaire (il serait plus facile de supporter un autre type de langage assembleur). Mais dans ce projet, nous réalisons les deux étapes en même temps.
Si vous travaillez en Python vous aurez à modifier le fichier `generation_code.py` et créer le fichier `table_symboles.py`. Si vous travaillez en C, vous aurez à modifier les fichiers `generation_code.c` et `generation_code.h` et créer les fichiers `table_symboles.c` et `table_symboles.h`.
Je vous conseille de commencer la génération de code même si vous n'avez pas totalement fini l'analyse syntaxique et de revenir à l'analyse syntaxique quand vous serez bloqué.

2 Si vous n'arrivez pas à exécuter les fichiers .nasm

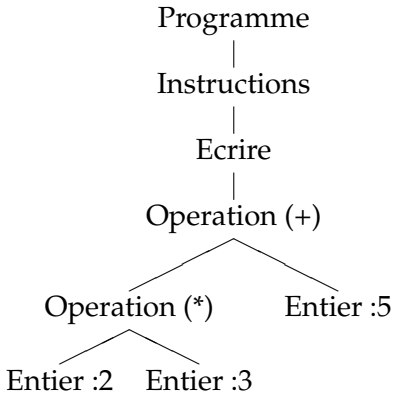
Si après avoir suivi les instructions d'installation, vous n'arrivez pas à compiler les fichiers assembleurs .nasm pour les exécuter (notamment si vous travaillez sur mac), vous pouvez utiliser ce compilateur en ligne de fichier .nasm.
Pour ceci, dans votre makefile, supprimez les instructions permettant de compiler les fichiers .nasm (`nasm ... ;` et `ld ... ;`) et l'instruction qui supprime les fichiers .nasm (`rm output/$${a}.nasm;`). Vous pouvez copier/coller le code de vos fichiers .nasm directement dans l'onglet `Source Code` et cliquer sur `Execute`.
Le problème principal est que ce site ne permet pas d'ajouter le fichier `io.asm` dont on a besoin pour gérer les entrées/sorties.
Solution : on supprime la première ligne `%include "io.asm"` des fichiers .nasm et on la remplace par le contenu du fichier. Pour ça, par exemple, on peut commenter la ligne qui créer cet include dans votre fichier `generation_code.py` ou `generation_code.c` et on ajoute dans le makefile une instruction
`cat io.asm output/$${a}.nasm > tmp; mv tmp output/$${a}.nasm;`.

3 Génération récursive de code

Étudions comment le compilateur fonctionne sur un fichier `exemple.flo` contenant seulement l'instruction :

```
ecrire(2 * 3 + 5);
```

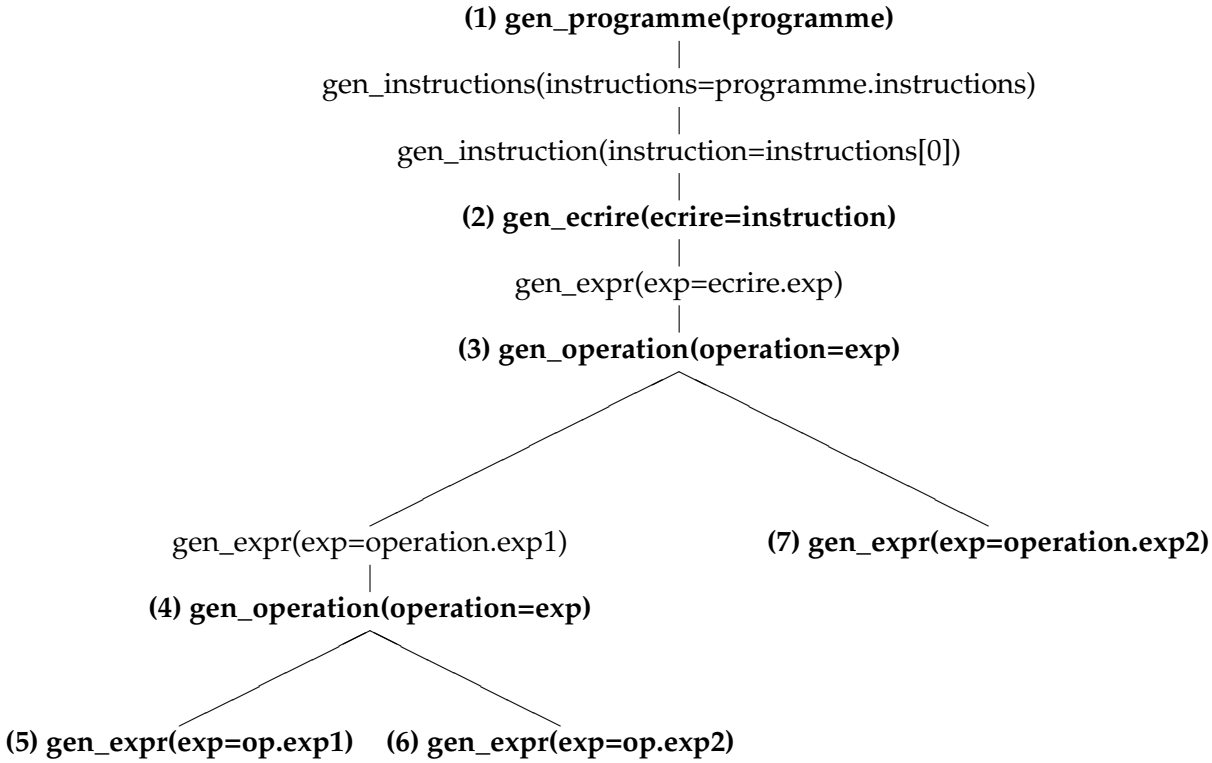
Si vous avez bien réalisé votre analyse syntaxique, ce programme est interprété comme un arbre abstrait :



Et voici le code correspondant généré (sans les commentaires) :

```
%include        "io.asm"
section .bss
sinput: resb    255     ;
v$a:    resd    1
section .text
global _start
_start:
        push    2
        push    3
        pop     ebx ;
        pop     eax
        imul    ebx ;
        push    eax
        push    5
        pop     ebx
        pop     eax
        add     eax, ebx
        push     eax
        pop     eax
        call    iprintLF
        mov     eax, 1
        int     0x80
```

Mais comment ce code a été généré ? Il est généré de manière récursive en appelant de la fonction `gen_programme(programme)` (ou son équivalent en C). Voici l'arbre des appels récursif (en **gras** et avec un numéro, les appels de fonction qui affichent du code) :



(1) La fonction `gen_programme` du fichier `generation_code.py` est appelée sur la racine de l'arbre (qui est un nœud de type `Programme`).

Cet appel affiche le code

```
%include        "io.asm"
section .bss
sinput: resb    255
v$a:    resd    1
section .text
global _start
_start:
```

qui correspond à un entête identique pour tous les programmes (pour le moment), lance la fonction `gen_instructions(programme.instructions)` puis écrit les instructions :

```
mov     eax, 1
int     0x80
```

qui ferme le programme.

(2) L'appel `gen_ecrire(ecrire)` prend en entrée le noeud `Ecrire` de l'arbre abstrait. Cette fonction va d'abord appeler la fonction `gen_exp(ecrire.exp)` qui va générer un code pour empiler la **valeur de l'expression** enfant de `Ecrire` dans l'arbre abstrait. Puis elle va écrire le code

```
pop     eax
call    iprintLF
```

`pop eax` permet de dépiler dans le registre `eax`. `iprintLF` est une procédure définie dans `io.asm` qui permet d'afficher un entier stocké dans le registre `eax`. Donc, ces deux instructions ensemble affichent la valeur qui était en haut de la pile.

(3) Le premier appel de `gen_operation(operation)` prend en entrée le nœud `Operation (+)` de l'arbre abstrait. Cette fonction va d'abord appeler la fonction `gen_exp(operation.exp1)` qui va générer un code pour empiler la **valeur de l'expression** enfant gauche de `Operation(+)` dans l'arbre abstrait, puis `gen_exp(operation.exp2)` pour son enfant droit. Ensuite il affiche le code suivant :

```
pop     ebx
pop     eax
add     eax, ebx
push    eax
```

Ce code dépile la valeur de l'enfant droit (le dernier empilé) dans `ebx`, la valeur de l'enfant gauche (le premier empilé) dans `eax`. Puis il utilise l'instruction `add` qui effectue l'opération `eax= eax+ ebx` puis il empile `eax`.

(4) Le second appel de `gen_operation(operation)` prend en entrée le noeud `Operation (*)` de l'arbre abstrait. Cette fonction appelle la fonction `gen_exp(operation.exp1)` qui va générer un code pour empiler la **valeur de l'expression** enfant gauche de `Operation(*)` dans l'arbre abstrait. Puis elle appelle `gen_exp(operation.exp2)` pour son enfant droit. Ensuite, elle affiche le code suivant :

```
pop     ebx
pop     eax
imul    eax, ebx
push    eax
```

Ce code dépile donc la valeur de l'enfant droit (le dernier empilé) dans `ebx`, la valeur de l'enfant gauche (le premier empilé) dans `eax`. Il utilise l'instruction `imul` qui effectue l'opération `eax= eax* ebx` (multiplication signéd) puis il empile `eax`.

(5,6,7) Les 3 appels `gen_expr(expr=...)` prennent en entrée les noeuds `Entier(2)`,`Entier(3)` et `Entier(5)`. Elles vont générer le code

```
push 2
```

qui empile la valeur 2 sur la pile (et même chose pour 3 et 5)

4 Opérations arithmétiques

Pour le moment, votre compilateur n'implémente que les opérations d'addition et multiplication. Vous trouverez en dernière page un résumé des différentes opération possible en x86 qui pourraient vous être utiles.

★ **1.** Modifier la fonction `gen_operation(operation)` (ou son équivalent C) pour supporter la soustraction, la division entière (sur des entiers signés) et l'opération modulo.

Selon comment vous avez défini l'opération unaire `- expr`, vous pouvez également l'implémenter ici.

5 Expression : Lire

On va maintenant implémenter l'entrée `lire()` qui : met en pause le programme, permet à l'utilisateur d'entrée au clavier une chaîne de caractère qui est interprétée comme un entier. On peut se servir du code suivant :

```
mov     eax, sinput
call    readline
call    atoi
push    eax
```

La première ligne charge l'adresse `sinput` (une zone mémoire de 255 octets que l'on a définis au début du programme) dans `eax`. La seconde ligne appelle la procédure `readline` de `io.asm` qui copie l'entrée utilisateur à l'adresse indiquée dans `eax` (donc `sinput`). Remarque : la procédure `Readline` ne modifie pas `eax`. La troisième ligne appelle la procédure `atoi` de `io.asm` qui transforme la chaîne de caractère à l'adresse indiquée dans `eax` en entier et met le résultat dans `eax`. La dernière ligne empile `eax`.

★ **2.** Implémenter le nouveau type d'expression `lire()`.

6 Booléens

On peut représenter en mémoire un booléen comme un entier qui vaut 0 pour faux et 1 pour vrai.

★ **3.** Sur le modèle de ce qui existe déjà pour les entiers, faire la génération de code des expressions qui correspondent aux booléens `Vrai` ou `Faux`.

Maintenant `ecrire(Vrai)` affiche 1 et `ecrire(Faux)` affiche 0.

7 Opérateurs logiques

On voudrait implémenter la génération de code pour les 3 opérations logiques sur les booléens : `ou`, `et` et `non`. Sur la dernière page, vous pouvez trouver les opérations x86 utiles. Attention : pour faire la négation, vous ne voulez probablement pas utiliser l'opération x86 `not` qui fait la négation bit à bit. Vous voulez probablement utiliser l'opération `xor` qui fait le xor bit à bit.
Après avoir implémenté les opérateurs logiques, vous allez être confronté à un problème : il ne devrait pas être permis d'utiliser un opérateur logique sur un entier. Par exemple `ecrire(non 5)` n'est pas un code valide et doit donner lieu à un message d'erreur. Autrement dit, on doit faire une vérification des types.
Comment procéder ? On peut vérifier la cohérence d'une expression et déterminer son type de manière récursive. Tout d'abord un nœud `Entier` ou `Lire` est et de type `entier`. À l'inverse, un nœud `Booléen` est de type `booléen`. Une opération arithmétique a deux enfants qui **doivent être de type entier** et est elle-même de type `entier`. De la même façon, un opérateur a des enfants qui **doivent être de type booléen** et est lui-même de type booléeen.
Comment indiquer le type ? Il y a plusieurs solutions. On pourrait ajouter à toutes nos expressions un attribut "type" qui pourrait être `entier` ou `booléen`. Sinon, on pourrait faire en sorte que la fonction `gen_expr` renvoie le type de l'expression dont elle vient de générer le code.

★ **4.** Implémenter la génération de code des opérateurs logiques et faire la vérification de type.

Maintenant `ecrire(Vrai ou Faux);` affiche 1 et `ecrire(Non 5);` affiche un message d'erreur à la compilation.

8 Comparaisons

On voudrait maintenant être capable de traiter les 6 opérations de comparaison. Quand on fait l'opération de comparaison `exp1 == exp2` on vérifie que `exp1` et `epx2` sont bien de type entier (et le résultat de la comparaison est de type booléen).
Il n'existe pas en x86 une opération élémentaire qui permet de récupérer 0 ou 1 selon si deux variables sont égales ou différentes. On peut quand même s'en sortir en utilisant les sauts conditionnels et les étiquettes.
Voici un exemple de saut conditionnel :

```
1         cmp     eax, ebx
2         je      e0
3         add eax ebx
4         jmp     e1
5 e0:
6         push ecx
7 e1:
```

Premièrement, aux lignes 5 et 7, on peut voir `e0:` et `e1:`. `e0` et `e1` sont des étiquettes (*label* en anglais). Une étiquette donne un nom à une ligne et permet donc de désigner cette ligne quand on veut "y sauter". Par exemple, à la ligne 4, `jmp e1` signifie que l'on saute (de manière inconditionnelle) à l'étiquette `e1` (= à la ligne 7). À la ligne 1, l'instruction `cmp eax, ebx` compare `eax` et `ebx`. Le résultat est stocké dans un registre spécial. À la ligne 2, `je e0` indique que l'on saute à l'étiquette `e0` si le résultat de la comparaison était nul : autrement dit, si `eax == ebx`. Globalement, ce programme met `eax+ebx` dans `eax` si `eax=ebx` et empile `ecx` sinon.
En vous inspirant de ce code, vous pouvez créer toutes les opérations de comparaisons. Encore une fois, se reporter à la dernière page pour trouver la listes des différents sauts conditionnels.
Bien sûr, toutes les étiquettes de votre programme doivent avoir un nom différent. Vous pouvez appeler toutes vos étiquettes `ei` en remplaçant i par un numéro de plus en plus grand. Pour ça vous pouvez écrire une fonction qui vous donne le nom de la prochaine étiquette.
Par exemple en Python :

```python
num_etiquette_courante = -1
def nom_nouvelle_etiquette():
        global num_etiquette_courante
        num_etiquette_courante+=1
        return "e"+str(num_etiquette_courante)
```

Ou en C :

```c
int num_etiquette_courante = 0;
void nouveau_nom_etiquette(char *etiq) {
  sprintf(etiq, "e%d", num_etiquette_courante++);
}
```

★ **5.** Implémenter la génération de code des opérateurs de comparaison.

Maintenant, `ecrire(3*2 != 6*1);` affiche 0 et `ecrire(7==4+3);` affiche 1.

9 Boucles et instructions conditionnelles

Avec les sauts conditionnels, on peut implémenter facilement les instructions de type boucle ou conditionnels. Pour une boucle il faut procéder comme suit :

(1) On évalue l'expression de condition. L'expression doit être un booléen.
(2) Si la condition vaut vrai alors on exécute la liste d'instructions Après avoir exécuté la liste d'instructions, on retourne à la première étape (saut inconditionnel).
(3) Si la condition est fausse alors on saute après la liste d'instructions.

Les instructions conditionnelles (`si, sinon si, sinon`) fonctionnent de manière un peu similaire.

★ **6.** Implémenter la génération de code des opérateurs de comparaison.

10 Fonctions sans arguments

Pour implémenter la génération de code des fonctions ou des variables, il faut d'abord implémenter une *table des symboles*.
Pour commencer, on va supposer que les fonctions définies n'ont pas d'arguments. Pour le moment, votre table des symboles permet simplement d'associer à chaque nom de fonction son type (entier ou booléen).

★ **7.** Créer une classe (ou une structure de donnée) qui représente votre table des symboles. Vous pouvez l'ajouter dans un nouveau fichier `table_des_symboles.py` (ou `.c et .h` si vous travaillez en C). Si nécessaire, modifier le Makefile.

Voici un exemple de programme et la table des symboles associée :

```
entier f(){
        retourner 2;
}
entier g(){
        retourner 4;
}
booleen h(){
        retourner Vrai;
}
ecrire(f()+g());
ecrire(Non h());
```

Nom fonction	Type
f	entier
g	entier
h	booléen

Le code suivant montre pourquoi il est important d'ajouter toutes les fonctions à votre table de symbole **avant** de commencer la génération.

```
entier g(){
 retourner h();
}
entier h(){
 retourner 3;
```

```
}
ecrire(g());
```

En effet, quand dans la fonction g, la fonction h est appelée, il faut que le compilateur vérifie que la fonction h existe belle et bien.

★ **8.** Avant de générer le code (par exemple, dans votre fonction `gen_programme(programme)`), parcourir la liste des fonctions pour les ajouter dans la table des symboles.

On veut maintenant générer le code des différentes fonctions. Vous pouvez créer une fonction `gen_def_fonction(f)` (ou équivalent) qui génère le code correspondant à la fonction `f`.
En `nasm`, vous pouvez coder une fonction grâce à une `procédure`.
La procédure commence par une étiquette qui peut être le nom de la fonction précédé d'un *underscore*. Le reste du code est simplement le code correspondant à la liste des instructions de la fonction.
On implémentera ensuite deux types d'instructions :

- `retourner expr` qui évalue l'expression puis met sa valeur dans `eax` avant de revenir à l'appel de la fonction grâce à l'instruction `nasm ret;`
- l'appel de fonction qui pour le moment, saute dans la procédure correspondant à la fonction grâce à l'instruction `nasm call`.

L'appel de fonction peut être une expression du type de celui de la fonction comme dans le code suivant :

```
entier f(){
 retourner 3;
}
ecrire(f());
```

Mais l'appel de fonction peut aussi être une instruction. Dans ce cas-là on ignore le résultat (et on n'empile pas `eax`), comme dans le code ci-dessous :

```
entier f(){
 ecrire(120);
 retourner 0;
}
f();
```

Voici un exemple simple de code et un équivalent possible en `nasm` :

```
entier f(){
 retourner 3;
}
ecrire(f());
```

```
%include        "io.asm"
section .bss
sinput: resb    255
v$a:    resd    1
section .text
global _start
_f:
        push    3;
        pop     eax
        ret
_start:
        call    _f
        push    eax
        pop     eax
        call    iprintLF
        mov     eax,    1
        int     0x80
```

★ **9.** Implémenter la génération du code des fonctions et les instructions (ou l'expression) d'appel de fonctions et de retour de fonctions.

En `FLO`, il est interdit de faire une instruction `retourner ...` en dehors d'une fonction. Par exemple le programme suivant ne doit pas compiler :

```
entier f(){
        retourner 3;
}
ecrire(f());
retourner 5;
```

De même, une fonction de type `entier` ne doit retourner que des entiers et une fonction de type `booléen` ne doit retourner que des booléens.
Par exemple les 2 programmes suivants ne doivent pas compiler :

```
booleen f(){
        retourner 10;
}
ecrire(f());
```

```
entier f(){
        retourner Vrai;
}
ecrire(f());
```

Solution : dans la table des symboles (ou en variable globale), on peut mémoriser la fonction dont on génère le code.
Si on est en dehors de toute fonction, l'instruction `retourner` génère une erreur de compilation. Si on est dans une fonction f de type t et qu'on effectue un retour, alors on vérifie que ce retour est de type t.

★ **10.** Faire en sorte que les 3 programmes ci-dessus ne compilent pas.

11 Fonctions avec arguments

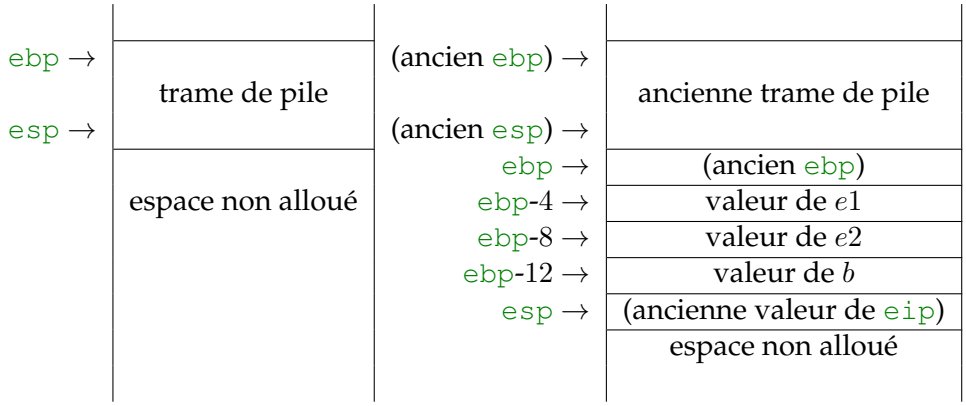
On va maintenant autoriser les fonctions à avoir des arguments.
La table des symboles va devoir stocker de nouvelles informations sur chaque fonction : le nombre et le type de ses arguments et la mémoire nécessaire pour ses arguments et variables locales. Pour le moment la mémoire nécessaire est seulement le nombre d'arguments de la fonction multiplié par 4.
Voir le programme suivant et sa table de symbole associé :

```
booleen f(entier e1, entier e2){
        retourner e1 > e2;
}

entier g(booleen b1, entier e2){
        si ( b1) {
                retourner e2;
        }
        retourner -e2;
}

entier h(entier e1, entier e2, entier e3){
        retourner e1 + e2 * e3;
}

ecrire(f(1,2));
ecrire(g(Faux,4));
ecrire(h(1,2,3));
```

Nom fonction	Type	mémoire	arguments
f	booléen	2*4=8	[entier,entier]
g	entier	2*4=8	[booléen,entier]
h	booléen	3*4=12	[entier,entier,entier]

★ **11.** Ajouter les types des arguments et la mémoire nécessaire pour la fonction dans la table des symboles. Maintenant, quand on fait un appel de fonction, on doit vérifier à l'aide de la table des symboles que le nombre et le type des arguments passés en paramètre est bon. En cas d'incohérence, il faut faire une erreur de compilation.

Maintenant que les fonctions nécessitent de la mémoire, comme on l'a vu en cours, les appels de fonctions vont créer des trames de pile.
Par exemple, l'appel suivant `f(1,2,Vrai)` de la fonction `entier f(entier e1,entier e2,booleen b)` va changer la pile comme suit :



(Ou de manière équivalente vous pouvez faire en sorte que `ebp` pointe sur la valeur de $e1$ et pas sur l'ancienne valeur de `ebp`. Ceci vous permet de faire en sorte que la première valeur soit stockée en [`ebp`] et pas en [`ebp`-4].)
Maintenant, quand on appelle une fonction il faut :

- **(a)** empiler la valeur de `ebp` avant de la changer (ancien `esp`-4) (ou -8).
- **(b)** empiler les arguments (ce qu'on peut faire en évaluant les expressions)
- **(c)** empiler le pointeur de programme `eip`
  (opération effecutée par `call`),
- **(d)** aller à l'adresse de la fonction
  (opération effecutée par `call`),

A l'issue de l'appel, il faut :

- **(a)** désallouer la mémoire de la fonction (en augmentant la valeur de `esp`) ,
- **(b)** rétablir la valeur de `ebp`.
- **(c)** empiler le retour de la fonction si l'appel de fonction était une expression. Sinon rien.

★ **12.** Modifier l'appel des fonctions pour qu'il génère une nouvelle trame de pile.

On voudrait maintenant implémenter la lecture de variable pour par exemple que le programme suivant affiche 5 :

```
entier f(entier e){
        ecrire(e);
        retourner 1;
}
f(5);
```

Pour ça, on doit pouvoir ajouter des variables dans la table des symboles pour que le programme sache où aller chercher la valeur de e. Une variable dans la table des symboles est caractérisée par un nom, un type (entier,booléen) et une adresse (relative à `ebp`). On lui ajoutera une portée plus tard quand on parlera des variables locales, mais un argument est accessible dans toute la fonction. Pour le moment, on doit seulement vider la table des symboles quand on quitte la définition d'une fonction.

★ **13.** Modifier la table des symboles pour permettre d'ajouter des variables. Au début de la génération de code d'une fonction, ajouter tous ses arguments comme variables. À la fin de la génération de code, supprimer toute les variables.

On peut maintenant implémenter les expressions de type variable en accédant à la variable `[ebp - adresse_de_e]`.
Par exemple la fonction *f* :

```
entier f(entier e){
        ecrire(e);
        retourner 1;
}
```

peut être compilée comme ceci :

```
_f:
        mov     eax,    [ebp - 4] ; met la valeur de la variable e dans eax
        push    eax
        pop     eax
        call    iprintLF
        push    1
        pop     eax
        ret
```

★ **14.** Implémenter les expressions de type variable. Le type de l'expression de type variable est déterminée par le type de la variable indiquée dans la table des symboles.

À ce stade, le programme suivant :

```
entier fibo(entier n){
 si(n<=1){
  retourner 1;
 }
 retourner fibo(n-1)+fibo(n-2);
}

ecrire(fibo(0));
ecrire(fibo(1));
ecrire(fibo(2));
ecrire(fibo(3));
ecrire(fibo(4));
ecrire(fibo(5));
ecrire(fibo(6));
ecrire(fibo(7));
```

doit produire le résultat suivant :

```
1
1
2
3
5
8
13
21
```

Maintenant, on veut implémenter les affectations. Les affectations ne sont pas plus dures que les accès aux variables (toujours via [eax-adress_de_la_variable]). Cependant, il faut bien vérifier que le type de la variable est compatible avec le type de l'expression qu'on essaye de lui affecter.
Par exemple la fonction f :

```
entier f(entier e){
        e = 7;
        ecrire(e);
        retourner 5;
}

f(5);
```

peut être compilée comme ceci :

```
_f:
        push    7;      e=7
        pop     eax; e=7
        mov     [ebp - 4],      eax ; e=7
        mov     eax,    [ebp - 4]; ecrire(e)
        push    eax; ecrire(e)
        pop     eax; ecrire(e)
        call    iprintLF; ecrire(e)
        push    5 ; retourner 5
        pop     eax ;   retourner 5
        ret     ;retourner 5
```

★ **15.** Implémenter les affectations.

12 Variables locales

Maintenant, on voudrait implémenter la définition de variables locales. Pour le moment, nous allons nous concentrer sur les variables locales à l'intérieur des fonctions.
Il y a deux types de définition de variables :

- les définition-attributions `type nom = expr;` où on ajoute une variable de nom `nom` et de type `type` dans la table des symboles et où on associe l'expression `expr` à cette variable. On en profite pour vérifier que `expr` est bien du bon type.
- les définitions simples `type nom;` qui sont en faite des définitions-attributions où la valeur de l'expression est la valeur par défaut associée au type (0 pour les entiers, Faux pour les booléens).

Quand on ajoute une variable dans un bloc d'instructions (à l'intérieur d'un si, d'une boucle ou d'une fonction), cette variable ne doit pas être visible en dehors de ce bloc d'instructions. Par exemple, les 3 programmes suivant ne doivent pas compiler :

```
entier f(booleen b){
 si (b){
        entier a = lire();
        ecrire(a);
 }
 ecrire(a+5);
```

```
 retourner 1;
}

f(Vrai);

entier f(){
 entier a;
}
entier g(){
 a=5;
}

f();

entier f(booleen b){
 tantque (b){
  entier rep = lire();
  si ( rep > 10){
   b = Faux;
  }
 }
 retourner rep;
}

f(Vrai);
```

Une solution est que, pendant le parcours de l'arbre abstrait, la table des symboles retienne le nombre de blocs d'instructions imbriqués dans lequel le nœud courant se trouve.

- — On commence avec une profondeur $p = 0$.
- — Quand on entre dans un nouveau bloc, on incrémente la profondeur ($p = p + 1$).
- — Quand on définit une nouvelle variable x, on lui associe la profondeur p courante. L'adresse de x est l'adresse de la plus grande adresse de la table des symboles $+4$ (selon comment vous avez définie l'adresse, ça peut être 4 + le nombre de variables de votre table des symboles $*4$).
- — Quand on sort d'un bloc, on retire toutes les variables de profondeur p de la table des symboles. Puis on décrémente p ($p = p - 1$).

Voici la table des symboles pour un programme exemple :

```
0:
1:  entier a = lire();
2:  si ( a > 2){
3:   entier b = a*a;
4:   si (b > 3 * a){
5:    entier c = 25;
6:   }
7:   sinon {
8:    entier d = 25;
9:   }
10: }
```

Ligne 0 :

nom	type	adresse	profondeur ($p = 0$)

Ligne 1-2 :

nom	type	adresse	profondeur ($p = 0$)
a	entier	4	0

Ligne 3-4 :

nom	type	adresse	profondeur ($p = 1$)
a	entier	4	0
b	entier	8	1

Ligne 5 :

nom	type	adresse	profondeur ($p = 2$)
a	entier	4	0
b	entier	8	1
c	entier	12	2

Ligne 6-7 :

nom	type	adresse	profondeur ($p = 1$)
a	entier	4	0
b	entier	8	1

Ligne 8 :

nom	type	adresse	profondeur ($p = 2$)
a	entier	4	0
b	entier	8	1
d	entier	12	2

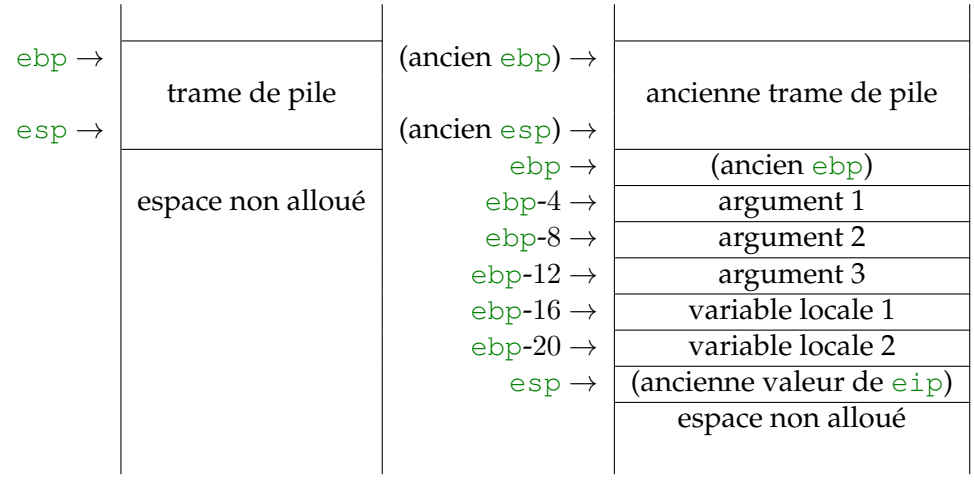
Ligne 9 :

nom	type	adresse	profondeur ($p = 1$)
a	entier	4	0
b	entier	8	1

Ligne 10 :

nom	type	adresse	profondeur ($p = 0$)
a	entier	4	0

★ **16.** Implémenter la définition de variables locales dans la table des symboles.
Maintenant, quand on appelle une fonction, il faut réserver de l'espace non seulement pour ses arguments, mais aussi pour ses variables locales.



Pour ça, il faut, avant d'appeler la fonction, diminuer la valeur de `esp` par l'espace occupée par les variables locales (soit l'espace pris par toutes les variables de la fonction - l'espace pris par les arguments).

Quand on parcourt une fonction f et qu'on ajoute une variable dans la table des symboles, si on voit que l'espace pris par toutes les variables dépasse la mémoire attribuée à f dans la table des symboles, alors on met à jour cette mémoire dans la table des symboles.
Cette modification permet de compiler le programme suivant par exemple.

```
entier puissance(entier a, entier b){
        entier i;
        entier r = 1;
        tantque( i < b){
                i=i+1;
                r = r * a;
        }
        retourner r;
}

ecrire(puissance(lire(),lire()));
```

Le problème vient quand une fonction s'appelle elle-même ou une fonction définie plus tard. Exemple :

```
entier somme(entier n){
        si (n == 0){
                retourner 0;
        }
        entier res = somme(n-1)+n;
        retourner res;
}
ecrire(somme(lire()));
```

Quand on fait le premier appel de `somme`, la table des symboles pense que `somme` n'a aucune variable locale. Il n'attribue donc pas suffisamment d'espace pour sa variable locale `res`.
Solution : Avant de générer du code, on fait un pré-parcours de toutes les fonctions pour mettre à jour la mémoire nécessitée par la fonction. Pour gagner du temps, on peut réutiliser les mêmes fonctions que la génération de code pour les fonctions en mettant temporairement la variable `afficher_nasm` (ou équivalent) à faux.

★ **17.** Implémenter la réservation d'espaces pour les variables locales de vos fonctions.

La toute dernière étape consiste à autoriser des variables locales pour le code en dehors des fonctions. Le problème rencontré est le même : il faut parcourir la liste des instructions, compter le nombre de variables locales utilisée simultanément au maximum et réserver l'espace nécessaire dans la pile.
Une solution efficace est de changer la structure du programme et de considérer le code en dehors de toute fonction comme une fonction `_main` particulière qu'on ajoute simplement à la liste de fonctions. Cette fonction ne prend pas d'argument, n'autorise pas l'instruction `retourner` et est lancée au début du programme.

★ **18.** Implémenter les variables locales en dehors des fonctions.

Liste partielle d'opérations Intel x86

Le processeur possède 4 registres de 32 bits d'usage général : `eax`, `ebx`, `ecx` et `edx`. En général, les instructions ont la forme `opcode dest, source` avec le code de l'opération suivi de la destination et de la source. Plusieurs modes d'adressage sont possibles pour la destination et la source, dont les noms de registres (`r`), les constantes (`imm`) et les adresses mémoire (`m`). La plupart des instructions n'acceptent pas deux arguments de type `m` en même temps.

`mov`	`r1|m1, r2|m2|imm`	Charge le deuxième argument dans le registre `r1` ou dans la position mémoire `m1`
`push`	`r|m|imm`	Charge l'argument sur le sommet de la pile
`pop`	`r|m`	Charge le sommet de la pile dans le registre `r` ou dans la position mémoire `m`
`add`	`r1|m1, r2|m2|imm`	Somme le deuxième argument au premier : `r1|m1 = r1|m1 + r2|m2|imm`
`sub`	`r1|m1, r2|m2|imm`	Soustrait le deuxième argument au premier : `r1|m1 = r1|m1 - r2|m2|imm`
`imul`	`r|m|imm`	Multiplie l'argument `r|m|imm` par `eax` et stocke le résultat dans `edx:eax`. Donc, `(edx:eax) = eax * r|m|imm`
`idiv`	`r|m|imm`	Divise `edx:eax` par l'argument. Met le quotient dans `eax` et le reste dans `edx`. Donc, `eax = (edx:eax) / r|m, edx = (edx:eax) % r|m` **Attention : penser à initialiser edx...**
`and`	`r1|m1, r2|m2|imm`	ET bit-à-bit du deuxième argument avec le premier : `r1|m1 = r1|m1 & r2|m2|imm`
`or`	`r1|m1, r2|m2|imm`	OU bit-à-bit du deuxième argument avec le premier : `r1|m1 = r1|m1 | r2|m2|imm`
`xor`	`r1|m1, r2|m2|imm`	XOR bit-à-bit du deuxième argument avec le premier : `r1|m1 = r1|m1` $\oplus$ `r2|m2|imm`
`not`	`r1|m1`	NON bit-à-bit de l'argument : `r1|m1 = ! r1|m1`
`cmp`	`r1|m1, r2|m2|imm`	Soustrait le deuxième argument au premier sans stocker le résultat : `r1|m1 - r2|m2|imm` (le résultat se voit dans les flags, voir ci-dessous)
`jl`	`e`	saut à l'adresse `e` si la flag SF $\neq$ OF (`r1|m1 < r2|m2|imm`)
`jg`	`e`	saut à l'adresse `e` si la flag SF = OF et SF $\neq 0$ (`r1|m1 > r2|m2|imm`)
`je`	`e`	saut à l'adresse `e` si la flag ZF $= 1$ (`r1|m1 = r2|m2|imm`)
`jle`	`e`	saut à l'adresse `e` si la flag SF $\neq$ OF ou ZF = 1 (`r1|m1` $\leq$ `r2|m2|imm`)
`jge`	`e`	saut à l'adresse `e` si la flag SF = OF (`r1|m1` $\geq$ `r2|m2|imm`)
`jmp`	`e`	saut inconditionnel à l'adresse `e`
`call`	`e`	saut inconditionnel à la procédure `e` avec sauvegarde de `eip`
`ret`		retour de procédure, reviens à la valeur sauvegardée de `eip`
`int`	imm	Interruption système ayant pour code `imm`, par exemple, `int 0x80` pour arrêter le programme