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

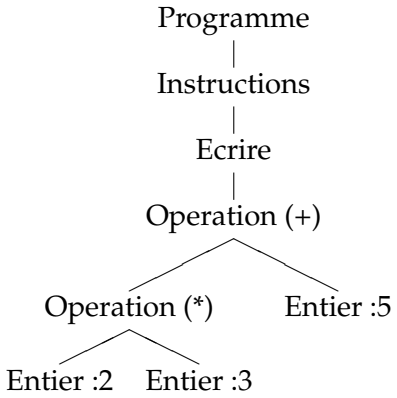
Si après avoir suivi les instructions d'installation, vous n'arrivez pas à compiler les fichiers assembleurs .nasm pour les exécuter (notamment si vous travaillez sur mac), vous pouvez utiliser ce compilateur en ligne de fichier .nasm.
Pour ceci, dans votre makefile, supprimez les instructions permettant 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);
```

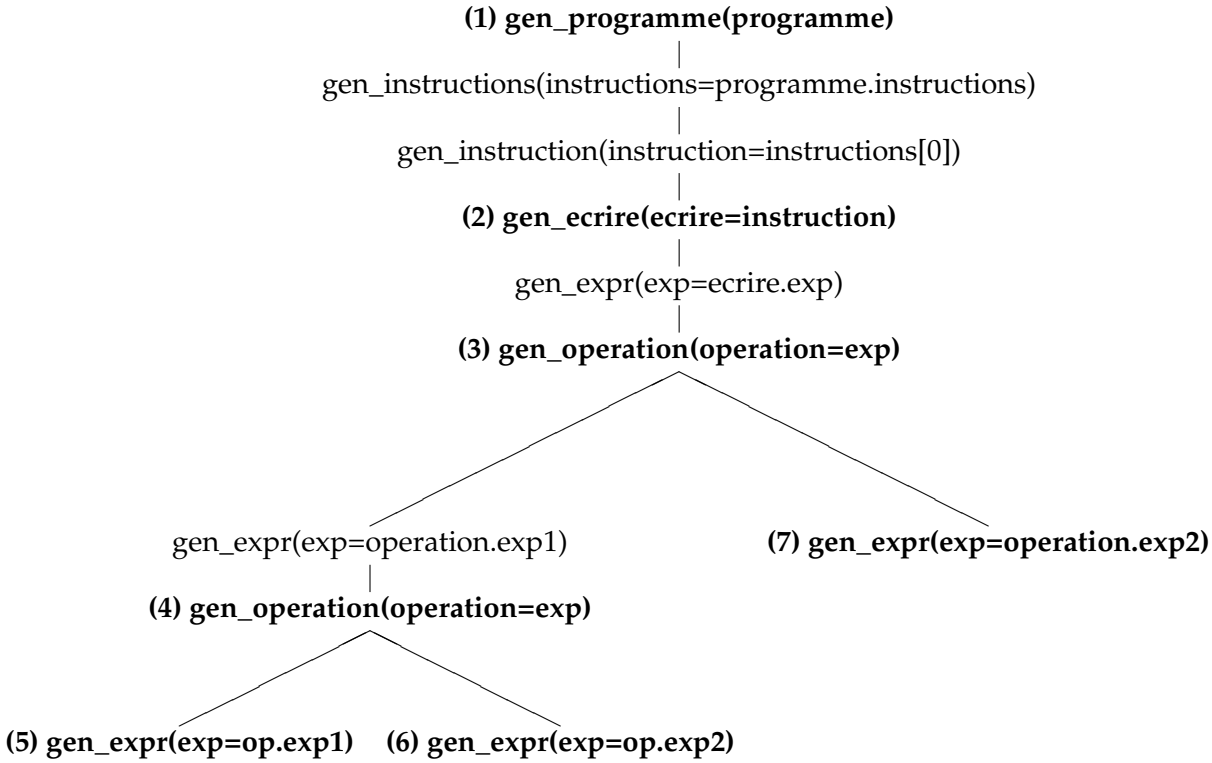
Si vous avez bien réalisé votre analyse syntaxique, ce programme est interprété comme un arbre abstrait :



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.

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