TP07 - GÉNÉRATION DE CODE - EXPRESSIONS, AFFECTATIONS, E/S

1. Objectif

L'objectif de ce TP est de commencer à générer du code X86. Nous nous limiterons ici à la génération de code pour les structures suivantes (dans l'ordre) :

(1) Déclarations de variables globales (simples et tableaux)

(2) Entrées (`lire`) et sorties (`ecrire`).

(3) Expressions comprenant des constantes, des variables globales et des opérateurs arithmétiques, logiques et de comparaison.

1.1. **Parcours.** La génération de code est réalisée lors d'un parcours de l'arbre abstrait, en même temps que les vérifications concernant la table des symboles (TP précédent).
Pour parcourir l'arbre abstrait, il faut au moins une fonction par type de nœud. On aura donc les fonctions `parcours_prog`, `parcours_instr`, `parcours_exp` et `parcours_dec`, et peut-être des fonctions spécifiques pour les différents types d'instructions, expressions et déclarations :

- `parcours_dec` : Pour le moment, les seules déclarations prises en compte sont les variables globales. Lors du parcours d'un nœud correspondant à une variable globale, il faut allouer de la place pour cette variable dans la région `.bss` du code machine, à l'aide des pseudo instructions `resb`, `resw` ... Pour l'instant, les déclarations de fonctions, de variables locales et de paramètres donnent lieu à des opérations dans la table des symboles, mais aucun code machine n'est généré.
- `parcours_instr` : Seules les instructions de type `ecrire` et `affect` seront traitées pendant ce TP. Une instruction `ecrire` doit être traduite par un appel système `int 0x80`, une affectation stocke la valeur d'un registre (qui contient le résultat de `parcours_exp`) dans une adresse mémoire. Les adresses des variables globales simples peuvent être représentées sous forme d'étiquette, tandis que les adresses des tableaux sont aussi des expressions dont la valeur doit être calculée.
- `parcours_exp` : Cette fonction génère les instructions correspondant au calcul de la valeur de l'expression et dépose le résultat du calcul dans la pile.

  La génération de code pour une opération de type `arg1 op arg2` consiste à affecter à des registres les deux opérandes `arg1` et `arg2` puis à réaliser l'opération à l'aide de l'instruction correspondante et enfin à empiler le résultat de l'opération.

  Pour les instructions complexes, les valeurs des opérandes (sous expressions) seront prises dans la pile, c'est là qu'aura été déposé le résultat de ces sous expressions. Il faut aussi traiter les lectures de variables `lire` avec un appel système.

1.2. **Point d'entrée.** Tout programme en L doit définir une fonction `main`, qui constitue le point d'entrée du programme. Le code produit par votre compilateur doit commencer par un appel à cette fonction. Nous verrons lors du TP prochain le détail de l'appel de fonction. Pour l'instant, il vous suffit de générer le code suivant :

```
global _start
_start:
call main
mov eax, 1  ; 1 est le code de SYS_EXIT
int 0x80  ; exit
main:
        ; on mettra ici le code généré
ret
```

1.3. **Entrées Sorties.** NASM n'offre que des procédures d'entrée sortie de bas niveau. Afin de disposer de procédures plus élaborées, en particulier pour la lecture et l'écriture des entiers, les procédures suivantes sont définies dans le fichier `io.asm` que vous trouverez sur la page web du site du cours.

- `readline` lit une ligne depuis l'entrée strandard et stocke le résultat dans une variable dont l'adresse se trouve dans le registre `$eax`. Attention, cette procedure ne gère pas le dépassement de tableau.
- `iprint` écrit l'entier contenu dans le registre `$eax` sur la sortie standard.
- `iprintLF` réalise le même traitement que la fonction précédente mais écrit un retour chariot après la valeur affichée.
- `sprint` affiche sur la sortie standard la chaîne de caractère dont l'adresse est contenue dans le registre `$eax`.
- `sprintLF` réalise le même traitement que la fonction précédente mais écrit un retour chariot après la valeur affichée.

NASM offre un mécanisme de macro qui permet en particulier d'inclure des fichiers de code. L'inclusion du fichier `io.asm` est réalisé de la façon suivante :

```
%include 'io.asm'
```

1.4. **Test.** Une fois que ces fonctionnalités basiques sont implémentées, il est possible de tester la génération de code sur des petits programmes contenant seulement la fonction main, des affectations, des expressions simples et des entrée-sorties, par exemple :

```
entier $a, entier $b;
main()
{
  $a = lire();
  $b = $a * $a;
  ecrire( $b );
}
```

Pour tester les programmes en assembleur `X86`, il faut les transformer en langage machine, à l'aide de l'outil `nasm` :

```
nasm -f elf -g -F dwarf test.asm -o test.o
```

puis faire l'édition de liens pour générer l'exécutable :

```
ld -m elf_i386 -o test test.o
```