TD8 : Génération de code (2)

Exercice 1

On s'intéresse à la génération de code pour l'expression suivante :

```
Exp = a (Op1 (b Op2 (c Op3 (d Op4 e))))
```

Question 1 Dessiner l'arbre correspondant à `Exp`.

Question 2 Donner le code généré par l'algorithme en une passe et k registres.

On s'intéresse maintenant à l'algorithme de génération de code en deux passes et k registres.

Question 3 Calculer `Nb_Reg` pour chaque nœud de l'arbre.

Question 4 Donner le code généré par l'algorithme en deux passes. Comment choisir les registres que l'on utilise pour stocker les expressions intermédiaires ?

Exercice 2

Soit la fonction factorielle (récursive cette fois!) :

```
int factorielle(int n) {
        if (n == 0) return 1;
        else return n*factorielle(n-1);
}
```

Question 1 Ecrire cette fonction dans notre langage d'assemblage.

Question 2 Ecrire cette fonction en langage d'assemblage sans utiliser l'instruction `CALL`.

Exercice 3

On reprend la grammaire du TD7, à laquelle on ajoute la possibilité de faire des appels de fonction. Un programme est donc désormais une liste de déclarations de fonctions, et une fonction peut avoir des paramètres :

prog	→	liste_decl `begin` liste_inst
function	→	`function (` liste_decl `)` liste_decl `begin` liste_inst
inst	→	place `:=` exp
inst	→	`call` Ident `(` Liste_Place `)`
Liste_Place	→	Place Liste_Place

On ajoute donc les sous-type d'arbre associés :

Programme	→	**Nœud_Programme**(Liste_Fonction)
Fonction	→	**Nœud_Fonction**(Liste_Decl,Liste_Decl, Liste_Inst)
Inst	→	**Nœud_Affect**(Place, Exp)
	|	**Nœud_Call**(Ident, Liste_Place)
Liste_Place	→	**Nœud_Liste_Place**(Place, Liste_Place)

Question 1 On suppose qu'on adopte la convention d'appel vue en cours. Ecrire la fonction `void Coder_Inst(Arbre A, [...])`, où `A` est un arbre qui dérive de Inst, qui produit du code pour les instructions (affectation ou appel de fonction).

Question 2 Modifier le code précédent de sorte a gérer le cas où la pile déborde.

Question 3 Écrire la fonction `void Coder_Fonction(Arbre A, [...])`, où `A` est un arbre qui dérive de Fonction, qui produit le code d'une fonction.

Exercice 4 Pour aller plus loin ... Compilation de langages orientés objet

En programmation, les structures (`struct` en C) permettent de rassembler un ensemble de valeurs de différents types et ainsi de composer des types plus complexes.

Question 1 Comment représente-t-on un élément de type `struct` dans la mémoire ?

En programmation orientée objet, on peut considérer qu'une classe est en fait une structure à laquelle on associe en plus des méthodes.

Question 2 Connaissant un objet, proposer un moyen d'appeler l'une des méthodes qui lui est associée.

Question 3 L'héritage est un concept important de la programmation orientée Objet. Une classe A peut donc hériter des attributs et des méthodes d'une classe B. Proposer un moyen d'appeler une méthode de la classe B à partir d'un objet de classe A.

Question 4 Que se passe-t-il si on cast un objet de type A en objet de type B ?

Langage d'assemblage utilisé

Opérandes :

- Adressage **immédiat** : `#42, #10, #1.2E-4...`
- Adressage **immédiat par registres** : `R0, R1, ..., LB, GB`
- Adressage **indirect avec déplacement** : `4(LB)` pour une variable locale, `1(GB)` pour une variable globale
- Adressage **indirect indexé avec déplacement** : `2(LB,4)` pour accéder à la 4ème valeur du tableau stocké à l'adresse `2(LB)`.

Les registres spéciaux pointant vers des adresses de la pile s'appellent `SP` (pointeur de sommet de pile), `LB` (base locale) et `GB` (base globale).

Instructions :

- `LOAD dval, Ri` : dval peut correspondre à tous les modes d'adressage, mais Ri doit nécessairement être un registre.
- `STORE Ri, dadr` : stocke la valeur stockée dans Ri à l'adresse dadr.
- `ADD dval, Ri` : ajoute la valeur stockée à l'adresse `dval` (notée `V(dval)`) au registre `Ri` (`Ri` ← `V(Ri) + V(dval)`).
- `SUB dval, Ri`
- `MUL dval, Ri`
- `DIV dval, Ri`
- `CMP dval, Ri` : mise à jour des codes condition selon `V(Ri) - V(dval)` Les codes conditions sont `EQ, NE, LT, LE, GT, GE`
- `BRA etiq` : branchement vers l'etiquette `etiq`
- `BEQ etiq` : branchement vers l'etiquette `etiq` si le code condition `EQ` est vrai
- `BNE etiq`, `BLT etiq`, `BLE etiq`, `BGT etiq`, `BGE etiq` : idem pour les autres codes condition
- `PUSH Ri` : stocke `Ri` en sommet de pile
- `POP Ri` : dépile et stocke dans `Ri`
- `HALT` : arrête le programme
- `WSTR str` : affiche la chaîne `str` : exemple `WSTR ''Hello World''`
- `WNL` : affiche un retour chariot
- `BOV` : branchement si l'opération précédente a fait un overflow
- `WINT` : affiche l'entier contenu dans `R1`
- `TSTO #n` : positionne le flag `OV` à la valeur `V[SP] + n > V[GB] + N`, où `N` est la taille de la pile.
- `CALL etiq` : permet d'appeler une fonction. A les effets de bord suivants : `SP` ← `V[SP]+2; V[SP]-1` ← `V[PC]; V[SP]` ← `V[LB]; LB` ← `V[SP]; PC` ← `V[etiq]`
- `RET` : retourne d'une fonction. A les effets de bord suivant : `PC` ← `V[V[LB]-1]; SP` ← `V[LB]-2; LB` ← `V[V[LB]]`