TP08 - GÉNÉRATION DE CODE (SUITE)

EXPRESSIONS (SUITE), INSTRUCTIONS, DÉCLARATIONS, APPELDE FONCTION

1. Objectif

L'objectif de ce TP est de compléter la génération de code machine à partir de l'arbre abstrait. Il faudra notamment traiter :

(1) les expressions avec opérateurs de comparaison et logiques,
(2) les instructions de contrôle (`si` et `tantque`),
(3) les instructions, expressions et déclarations liées aux fonctions
   (a) la déclaration de la fonction,
   (b) la liste de paramètres,
   (c) les déclarations de variables locales,
   (d) les appels de fonction dans les expressions,
   (e) les appels de fonction en tant qu'instruction,
   (f) la liste d'expressions,
   (g) l'instruction et la valeur de `retour`.

1.1. **Expressions (suite).** En L, il n'y a pas de distinction entre les expressions booléennes et arithmétiques. Ainsi, nous traitons les valeurs de toutes les expressions comme des nombres entiers, de la même manière qu'en C. Dans les tests conditionnels du `si` et du `tantque`, la valeur entière 0 représente *faux*, tandis que les valeurs différentes de zéro sont interprétées comme vrai.
L'évaluation des comparaisons `<` et `=` peut être effectuée avec des sauts conditionnels `jl` et `je`. Leur évaluation doit donner comme résultat la valeur 0 (*faux*) ou différent de 0 (*vrai*). Comme pour les autres expressions, le résultat sera empilé.
Les opérateurs logiques de L seront implémentés par des sauts et non par des instructions logiques en assembleur, qui effectuent des opérations logiques bit-à-bit, ce qui ne correspond pas à la sémantique des opérateurs logiques de L.
On ne cherchera pas à générer du code "court-circuit" pour les opérateurs logiques binaires `&` et `|`, qui consiste à ne pas évaluer la seconde opérande, dans le cas de l'opérateur `&`, si la première opérande est fausse (car on sait que le résultat sera faux) et à ne pas évaluer la seconde opérande, dans le cas de l'opérateur `|`, si la première opérande est vraie (car on sait que le résultat sera vrai).

1.2. **Instructions de contrôle.** Le langage L possède deux instructions de contrôle, le `si` et le `tantque`.

— L'instruction `si` *Exp* `alors` B_1 `sinon` B_2 est implémentée à l'aide de deux étiquettes : `sinon` et `suite`. L'étiquette `sinon` est l'adresse de la première instruction de B_2 (si l'instruction possède une partie `sinon`) et `suite` est l'adresse de la première instruction qui doit être exécutée après l'instruction `si`.
  Si le résultat de *Exp* est faux alors on saute vers `sinon`, si l'instruction possède une partie sinon. Si ce n'est pas le cas, on saute vers l'étiquette `suite`. Dans le cas où l'instruction possède une partie sinon, il faut veiller à sauter vers l'étiquette `suite` à la fin de B_1.
— L'instruction `tantque` *Exp* `faire` B est implémentée à l'aide des deux étiquettes `test` et `suite`. L'étiquette `test` est l'adresse de la première instruction de *Exp* et l'étiquette `suite` a le même sens que ci-dessus.
  Si le résultat de *Exp* est faux alors on saute vers `suite`. A la fin de B, on saute vers `test`.

	valeur de retour
	paramètre 1
	...
	paramètre k
	ancienne valeur de `eip`
`ebp`	ancienne valeur de `ebp`
	variable locale 1
	...
`esp`	variable locale n

FIGURE 1. Trame de pile

1.3. **Déclarations et appels de fonctions.** Un appel de fonction donne lieu à la création d'une trame de pile (ensemble des informations nécessaires à l'exécution de la fonction et à la poursuite de l'exécution du programme après l'appel). A l'issue de l'appel, la trame de pile doit être détruite. Une partie des opérations de création et de destruction de la trame de pile est réalisée par la fonction appelante. L'autre est réalisée par la fonction appelée (elle se trouve dans le code généré lors du parcours de la définition de la fonction).
Lors d'un appel de fonction, la fonction appelante doit :

(1) réserver de la place pour le résultat,
(2) empiler les arguments,
(3) empiler le pointeur de programme `eip` (opération effecutée par `call`),
(4) aller à l'adresse de la fonction (opération effecutée par `call`),

La fonction appelée doit :

(5) empiler le frame pointer `ebp`,
(6) initialiser la nouvelle valeur de `ebp`,
(7) réserver de la place dans la pile pour les variables locales.

A l'issue de l'appel, la fonction appelée doit :

(1) stocker le résultat,
(2) libérer la mémoire des variables locales,
(3) dépiler le frame pointer `ebp`,
(4) dépiler le pointeur de programme `eip` (opération effecutée par `ret`),
(5) sauter vers l'adresse de retour (opération effecutée par `ret`)

La fonction appelante doit :

(6) libérer la mémoire des arguments,
(7) dépiler le résultat,