

# Compilation — TD 5 : fonctions imbriquées

J. Boender & G. Henry

2009–2010

Dans les précédents TD nous nous sommes restreints à des fonctions *non-imbriquées* : les seules variables accessibles par une fonction étaient les variables globales, et ses variables locales. L'objet de ce TD est d'étudier la compilation des fonctions *imbriquées* : lorsque celle-ci peuvent accéder aux variables locales des fonctions qui les englobent. Sauf mention contraire, on se place dans le cadre du langage CTigre, du schéma de compilation et des conventions d'appel du projet.

**Exercice 1** Parmi les langages de programmation que vous connaissez, lesquelles permettent la définition de fonction imbriquée ?

**Exercice 2** Le langage CTigre permet-il la définition de variable globale ?

**Convention d'appel (rappel)** La structure du bloc d'activation (*frame*) de chacune des fonctions d'un programme CTigre (y compris de la fonction principale) est la suivante :

*adresse plus grande*

\$fp →	Dernier argument de la fonction
	...
	Premier argument de la fonction ( <i>offset</i> = −1)
	Lien statique
\$sp →	Première variable locale ( <i>offset</i> = +1)
	...
	Dernière variable locale
	Place pour les temporaires allouées en pile
	Place pour les registres à sauvegarder
	... dont \$fp
	... dont \$ra

*adresse plus petite*

La première partie du bloc (jusqu'au lien statique) doit être allouée par la fonction appelante ; la seconde moitié doit être allouée par la fonction appelée ; au moment de l'appel le registre \$sp pointe sur le lien statique. Les registres \$s0 à \$s7, \$sp et \$fp doivent être préservés par un appel de fonction : leur valeur doit être la même avant et après l'instruction `jal`.

**Level/offset (rappel)** Le *level* d'une fonction et le niveau d'imbrication de la fonction de nissant l'identificateur. Par convention, le *level* du programme principal sera 1. L'*offset* d'une variable est sa position dans le bloc, compte en nombre de mots vers le bas à partir du registre \$fp. Ainsi, le lien statique occupe le mot d'offset 0; les *offsets* des variables locales commencent à partir de 1; et ceux des arguments décroissent à partir de -1.

**Exercice 3** On considère le programme CTigre suivant :

```
var i := 10 in
function g(x : int): int =
  var a := 3 in
    function f(y : int) : int = y * i in
      var h(z : int) : int = z / a in
        f(h(x)) + a
    in g(12)
```

1. Indiquez pour chaque fonction la valeur de l'attribut *level* correspondant, et, pour chaque variable, les valeurs des attributs *level* et *offset* correspondants.
2. Décrivez la structure du bloc d'activation du programme principal et de chacune des fonctions.
3. Décrivez l'évolution de la pile lors de l'exécution du programme, en supposant que \$sp vaut 2000 au début de l'exécution du programme.
4. Comment est compilé, en pseudo-code C, l'appel à la variable a dans le corps de la fonction h?  
Vérifiez sur l'état de la pile au moment où la fonction h est en cours d'exécution (question 3), que la suite d'accès mémoire effectuées par ce morceau de code est correcte.

**Exercice 4** Mêmes questions que l'exercice précédent en considérant le programme suivant et en particulier la variable x dans le corps de la fonction g :

```
var a := 3 in
function f(x : int, n : int) : int =
  function g(n : int) =
    if n = 0 then 1
    else x * g(n - 1)
  in g(n)
in f(a, 2)
```

**Exercice 5 – Niveau des fonctions appelés** Étant donné une fonction *f* de niveau *n*, quels sont les niveaux possibles pour les fonctions qui peuvent être appelées depuis le corps de la fonction *f* ?

**Exercice 6 – Analyse d'échappement** Certaines variables locales sont utilisées uniquement dans le corps de leur fonction de définition et ne sont pas utilisées dans une fonction imbriquée. On dit alors qu'elles ne s'échappent pas de leur fonction de définition. Il n'est pas nécessaire de calculer d'offset pour

ces variables. Distinguer parmi les programmes ci-dessus, les variables qui ne s'échappent pas? D'une manière plus générale dans quel cas cette distinction sera importante?

**Exercice 7 – Partage d'offset** Dans le corps d'une fonction, certaines variables locales ont des portées disjointes. Proposez un programme comprenant de telles variables. Est-il alors possible de re-utiliser le même *offset* pour ces variables?

**Exercice 8 – Compilation vers C** L'objectif du projet est d'écrire un compilateur générant directement une suite d'instruction assembleur à partir d'un programme CTigre. Certains compilateurs se contentent de générer un programme équivalent écrit dans un autre langage, par exemple C, puis de faire appel à un compilateur complet pour ce second langage. Sauriez-vous traduire les programmes précédents dans un langage ne permettant pas la définition de fonction imbriquée?

**Exercice 9 – Initialisation des attributs *level* et *offset*** La première étape obligatoire du projet est de calculer les attributs *level* et *offset* pour chaque variable du programme à compiler. Pour vous aider, le code fourni contient une définition de type d'AST décorée avec ces informations. C'est le type `Ast.attrexp`. Par rapport au type `Ast.rawexp` des AST produit par l'analyseur syntaxique les symboles :

```
{ représentant une variable doivent être remplacés par le type :
    type varpos =
        | Stack of int × int (* Allocation in the stack (level, offset) *)
        | Temp of Temp.temp (* Allocation in a temp *)
    type attrvar = { v_name : symbol; v_pos : varpos }
{ représentant une définition de fonction doivent être remplacés par le type :
    type attrfundef = { f_name : lbl; f_level : int; }
{ représentant un appel de fonction doivent être remplacés par le type :
    type attrfuncall =
        | Internal of attrfundef
        | Predefined of lbl
```

Quels type de parcours de l'AST est nécessaire pour réaliser cette transformation? Quels paramètres sont nécessaires pour réaliser ce parcours?

**Exercice 10 – Optimisation** Avec le schéma de compilation qui vous est demandé dans le projet, pour accéder à une variable de niveau 1 depuis une fonction de niveau  $n$ , il faut  $n$  indirections, c'est-à-dire  $n$  accès à la mémoire avant l'accès à la variable elle-même. Sauriez-vous faire en une seule indirection? Vous pourriez comparer votre solution à ce que la littérature anglaise nomme *display table*.