Lien statiques, Frames

- Liens statiques
  - les blocs d'activation
  - la difficulté avec la portée statique
  - une solution: les attributs *level* et *offset*
- Analyse d'échappement

Exécution des fonctions

L'exécution d'un programme CTigre qui comporte des fonctions peut se faire en utilisant la *pile de blocs d'activation* que nous avons utilisé pour l'éxécution de fonctions en assembleur.

On reviendra plus avant sur les caractéristiques de CTigre qui font en sorte qu'une telle pile est suffisante. Pour l'instant il suffira de remarquer que ce n'est pas toujours le cas (notamment, Scheme et Ocaml ne peuvent se satisfaire d'une machine à pile).

Conventions...

Rappel, dans ce cours nous assumons que:

- la pile croît vers le bas,
- SP pointe sur la dernière case utilisée de la pile
- l'organisation du bloc d'activation est la suivante

(le pourquoi sera plus clair avant):

adresse plus grande

	in-arg n
	...
	in-arg 1
$FP \rightarrow$	lien statique
	var. loc. 1
	...
	var. loc. k
	place pour les temporaires
	place pour FP
	place pour ra
	place pour les autres registres...
	place pour les arguments des fonctions
$SP \rightarrow$	qu'on appelle...

adresse plus petite

Appel d'une fonction, création d'un bloc sur la pile

Voyons comment se déroule un appel de fonction f[1] selon la convention MIPS.
Il y a une partie du travail qui est faite par l'appelant:

- l'appelant mets en place les m paramètres actuels de la fonction appelée f dans la partie de son bloc près de SP réservée pour cela (comme on a décidé que SP ne bougera pas pendant l'execution du code de l'appelant, dans le bloc de l'appelant il y a déjà la place nécessaire pour allouer tous les arguments...)

```
M[SP+1] <- arg 1
...
M[SP+n] <- arg n
```

- l'appelant met en place le lien statique ls de f

```
M[SP] <- ls
```

- l'appelant appelle la fonction f (instruction `CALL f`)

Et une partie du travail qui est fait par l'appelé...

[1] en supposant que chaque paramètre et variable occupe exactement une case mémoire.

Appel d'une fonction, création d'un bloc sur la pile

prologue f *alloue son bloc*, de taille `framesize` sur la pile

```
SP <- SP-framesize
```

- l'appelé sauvegarde la valeur de FP, à la position `frameoffset` dans le bloc...

  ```
  M[SP+framesize-frameoffset] <- FP
  ```

  et positionne FP

  ```
  FP <- SP+framesize
  ```

- f a reçu par l'appelant dans un registre spécial ret l'adresse de retour, elle peut le sauver dans son frame, si nécessaire.

calcul

on exécute le corps de f, le résultat est dans un registre spécial res

épilogue f *désalloue son bloc* et restaure les valeurs de SP et FP, ...

```
FP <- M[SP+framesize-frameoffset]
SP <- SP+framesize
```

et saute à l'adresse de retour:

```
JUMP ret
```

Un exemple

Considérons le programme suivant

```
let g(x:int): int =
    let a := 50 in
    let f(y:int,z:int):int = y*y+z
    in f(x,a)+a
in g(3)
```

et ignorons pour l'instant le lien statique

Évolution de SP et FP

A l'exécution, on instancie les variables locales des fonctions exécutées et l'évolution de la pile suit le niveau d'imbrication des fonctions.

$FP \rightarrow 1100$	var. globales
	...
1001	place pour
$SP \rightarrow 1000$	les arguments

mise en place params g

$FP \rightarrow 1100$	var. globales
	$\ldots$
1001	$x = 3$
$SP \rightarrow 1000$	lien statique

appel de g

1100	var. globales
	$\ldots$
1001	$x = 3$
$FP \rightarrow 1000$	lien statique
999	$a = 50$
998	1100 (vieux FP)
997	(vieux RA)
996	place
995	pour
$SP \rightarrow 994$	les arguments

g empile les args. de f

1100	var. globales
	$\ldots$
1001	$x = 3$
$FP \rightarrow 1000$	lien statique
999	$a = 50$
998	1100 (vieux FP)
997	(vieux RA)
996	$z = 50$
995	$y = 3$
$SP \rightarrow 994$	lien statique

appel de f dans g

1100	var. globales
	$\ldots$
1001	$x = 3$
1000	lien statique
999	$a = 50$
998	$1100(vieux FP)$
997	(vieux RA)
996	$z = 50$
995	$y = 3$
$FP \rightarrow 994$	lien statique
$SP \rightarrow 993$	1000 (vieux FP)

Évolution de SP et FP

f retourne

1100	var. globales
	$\ldots$
1001	$x = 3$
$FP \rightarrow 1000$	lien statique
999	$a = 50$
998	1100 (vieux FP)
997	(vieux RA)
996	$z = 50$
995	$y = 3$
$SP \rightarrow 994$	lien statique

g retourne

$FP \rightarrow 1100$	var. globales
	$\ldots$
1001	$x = 3$
$SP \rightarrow 1000$	lien statique

L'attribut *offset*

Pour pouvoir accéder à ses propres variables, le code machine produit par la fonction devra connaître la *position dans son propre bloc d'activation* de ces variables.

Pour cela il est important d'attacher à chaque variable locale un attribut, traditionnellement appelé *offset*, qui donne cette position et qui sera utilisé pour générer le code qui accédera à cette variable.

Si on assume[2] que toutes les variables locales sont mémorisée dans le bloc d'activation, avec la convention que l'on a fixé, cette valeur peut être calculée en suivant l'ordre des déclarations des variables locales: offset vaudra 1 pour la première déclaration, 2 pour la deuxième etc. (la position 0 est prise par le lien statique).

Pour les paramètres formels, qui se trouvent positionnés de l'autre coté de FP, on peut choisir un offset négatif.

Le problème de la portée statique

La notion de bloc, avec portée statique, implique qu'une fonction définie localement à un bloc peut avoir accès à toutes les définitions du bloc englobant, et de celui qui englobe celui-ci, etc.

Considérons l'exemple suivant (qui calcule le même résultat que le précédent)

```
let g(x:int): int =
    let a := 50
    in let f(y:int):int = y*y+a
       in f(x)+a
in g(3)
```

A l'exécution, f aura besoin d'accéder aussi à la valeur de la variable a, qui est locale à g.

Comment la trouver?

L'attribut *level* et le *lien statique*

Dans ce langage toute fonction f en exécution peut accéder (outre ses paramètres, ses propres variables et les variables globales) seulement aux variables définies dans les fonctions $g_1, \ldots, g_n$ qui l'englobent dans le texte du programme.
Ces fonctions ont un *niveau d'imbrication* inférieur à celui de f, et leur bloc d'activation est forcément sur la pile au moment de l'exécution de f (comme dans le cas de g qui englobe f dans l'exemple).

Nous pouvons associer à chaque fonction un attribut, traditionnellement appelé *level*, qui corresponde au niveau d'imbrication des fonctions.
Cet attribut sera associé ensuite à chaque variable locale de la fonction.

L'attribut *level* et le *lien statique*

Dans notre exemple, la fonction g, qui est définie à l'intérieur du programme principal, aura *level=2*, alors que f aura *level=3*. Donc la variable a locale à g aura *level=2*,

[2] La question est plus complexe si on garde une partie des variables dans des registres machine.

offset=2.

Maintenant, quand on compile la fonction f et on trouve la référence à la variable a, on connaît, en consultant la table des symboles, ces deux attributs.

Il ne nous reste qu'à écrire le code qui accède à la composante *offset* du *plus récent* bloc d'activation qui se trouve sur la pile et qui corresponde à une fonction *englobant* f de niveau *level*.

L'attribut *level* et le *lien statique*

Comment une fonction f peut retrouver le plus récent bloc d'activation qui se trouve sur la pile et qui corresponde à une fonction *englobant* f de niveau *level*?

Une solution simple consiste à passer à chaque fonction f, au moment de l'exécution, un pointeur vers le bloc d'activation de la fonction g *qui la définit* dans le programme. Ce pointeur est appelé le *lien statique*, et il s'ajoutera aux paramètres de la fonction.

Si nous sommes une fonction f de niveau k et nous cherchons à trouver le bloc d'activation d'une fonction g de niveau $l < k$, il nous suffira de suivre $k - l$ fois le lien statique pour le joindre.

Si nous suivons la convention de mettre toujours le lien statique en première position dans le bloc, si f cherche la variable de niveau l et offset o, elle la trouvera dans

$$M[\underbrace{M[\ldots M[}_{k-l\ fois} FP]\ldots] - o]$$

.

L'attribut *level* et le *lien statique*

Trouver la variable de *level* et *offset* donné est possible.

	$\ldots$
1001	$x = 3$
1000	lien st. = 1100
999	$a = 50$
998	$1100 (vieux FP)$
997	$y = 3$
$FP \rightarrow 996$	lien st. = 1000
$SP \rightarrow 995$	1000 (vieux FP)

bloc de g, level=2

a: level=2,offset=1

bloc de f, level=3

Pour f (niveau 3)[3], a (niveau 2, offset 1) est $M[M[fp] - 1] = M[999]$.

[3]Nous suivons la convention de mettre toujours le lien statique en première position dans le bloc

Un exemple complexe

```
type tree ={key: strinf, left: tree, right: tree} in
let pretty(tree:tree):string =
 let output := "" in
 let write(s: string) = output:=concat(output,s) in
 let show(n:int, t:tree) =
   let indent(s:string) = (for i=1 to n do write(" ") done;
                           output:=concat(output,s))
   in if t=nil then indent(".")
      else (indent(t.key);show(n+1,t.left);show(n+1,t.right))
 in show(0,tree); output
in pretty(nil)
```

l=2 l=3 l=3 l=4

Ici, il y a plusieurs cas intéressants:

- un appel normal d'une fonction par la fonction qui la définit: pretty appelle show et passe sont propre FP comme lien statique à show
- un appel récursif de show: là on passe comme lien statique à l'appel récursif *le lien statique* du show appelant
- un appel de la part d'une fonction imbriquée d'une fonction définie plus à l'extérieur: indent appelle write et doit lui passer comme lien statique le FP de pretty. Elle l'obtient en suivant les lien statiques jusqu'au lien statique passé à show.
- indent utilise output, définie dans pretty. Elle suit la chaîne statique pour ça

Table pour le calcul du lien statique à passer à l'appelé

niveau appelant	niveau appelé	lien statique à empiler	note
l	l+1	mon FP	
l	l	mon LS	= M[FP]
l	l-k	$\underbrace{M[\ldots M[}_{k fois} LS] \ldots]$	= $\underbrace{M[\ldots M[}_{k+1 fois} FP] \ldots]$

N.B.: au moment ou l'on compile un appel de fonction, on aura sous la main à la fois les informations sur l'appelant (que l'on est en train de compiler) et de l'appelé (dont étiquette et niveau seront déjà connus).

Les informations qui nous servent sur un bloc

A titre d'exemple, voici un extrait des déclarations de type pour un fragment dans mon implémentation du compilateur:

```
type frame = {
  entry: Temp.label; (* point d'entree, au debut du prologue *)
  mutable maxargs: int; (* nombre maximum d'arguments d'une fonction appele
  mutable numtemps: int; (* nombre de variables/temporaires sur la pile *)
  }
```

- `maxargs`: le nombre maximum de paramètres d'une fonction appelée
  on peut le calculer à plusieurs endroit dans la chaîne de compilation
  on vous suggère de le faire sur le code intermédiaire

- `numtemps`: le nombre de mots utilisés pour variables locales + temporaires
  on ne le connaîtra qu'au but[4] de la chaîne de compilation
  vous le calculerez juste avant d'émettre prologue et épilogue pour l'assembleur.
  Donc, il faudra garder l'information sur le bloc jusqu'au but!

Analyse d'échappement (escape analysis)

On divise les variables locales d'une fonction f en deux groupes:

variables utilisées seulement par f : ces variables locales peuvent, s'il y a la place, être mises dans des registres machines;

variables utilisées aussi en dehors de f : par exemple, la variable a du dernier exemple est locale à g, mais utilisée par f;
On dit que ces variables "échappent", et on est en général obligés de les allouer dans la pile, pour qu'on puisse leur donner une adresse utilisable par qui fait référence à ces variables en déhors de la fonction qui les a créés.
Dans des langages comme C, une variable dont on prend l'adresse, comme dans

```
void *breakit() {
        int i=100;
        . . .
        return(&i);
}
```

"échappe" aussi.

Approximation dans le projet

Dans le cadre de votre projet, on assumera que toutes les variables locales échappent, et donc on les allouera toutes en pile.
Cependant, pour un langage comme CTigre, on pourrait facilement procéder à une "analyse d'échappement": en visitant l'AST, on peut savoir si une variable sera ou pas utilisée en déhors de la fonction qui la définit.
On pourrait alors allouer en pile (en calculant level et offset) seulement les variables qui échappent.
Cela laisserait une chance à l'allocateur de registres de mettre quelques variables dans des vrai registres machine.

[4] même la séléction d'instructionscrée des temporaires!