Traduction des langages – Placement mémoire

Objectif :
- Définir la nouvelle structure d'arbre obtenue après la passe de placement mémoire
- Définir les actions à réaliser par la passe de placement mémoire

1 Placement mémoire

- **Lire les transparents 118 → 134.**
- **Le point important est qu'on va générer du code pour une machine à pile. Par exemple le code correspondant à "3 + 4" fait : empiler 3, empiler 4, appeler + (qui consomme ses deux arguments et empile le résultat). À venir dans le cours suivant sur la génération de code.**
- **La pile contient les variables locales du bloc principal, et pour chaque invocation de fonction, les paramètres et les variables locales de la fonction.**
- **Pour le bloc principal, les variables sont rangées par rapport à la base de la pile (SB, Stack Base) : la première variable en SB+0, la deuxième en SB+x où x est la taille de la première, etc.**
- **Noter que quand il y a des choix (if), les variables des deux blocs peuvent être rangées aux mêmes emplacements vu qu'une seule des deux branches sera exécutée ; et quand un bloc se termine, ses variables disparaissent et leur emplacement peut être réutilisé.**
- **Pour une fonction, LB (link base) est la base de l'enregistrement d'activation qui contient les informations nécessaires à cette invocation (adresse de retour notamment), cf transparent 128. Les paramètres sont sous l'enregistrement d'activation, et les variables locales au-dessus. (note avancée : on ne peut pas connaître, au moment de la compilation, l'emplacement absolu de ces variables et paramètres vu que ça dépend du chemin d'exécution ; penser en outre aux appels récursifs : il y a plusieurs invocations actives de la même fonction)**
- **À l'appel d'une fonction, les arguments sont empilés de gauche à droite, donc le *dernier* paramètre est le plus haut dans la pile, en LB-x (où x est sa taille), l'avant-dernier en LB-x-y (où y est la taille de l'avant-dernier), etc.**
- **Les variables locales sont, similairement à celles du bloc principal, en LB+3, LB+3+x etc (3 est la taille de l'enregistrement d'activation, les variables locales commencent au-dessus).**

▷ **Exercice 1** *Donner le placement mémoire des variables du programme suivant :*

```
rat f3 (int a int b rat r){
  return [(a + num r) / (b + denom r)];
}
rat f2 (bool b rat x rat y){
  int x1 = num x;
  int x2 = denom x;
  rat res = call f3(x1 x2 y);
  return res;
}
int f1 (int i rat r int n){
  rat r1 = call f2(true r [i/n]);
  return denom r1;
```

```
}
test{
  int x = call f1 (13 [4/11] 17);
  rat y = call f2 (true [4/11] [11/4]);
  rat z = call f3 (1 2 [4/11]);
  print ((x+denom y)+num z);
}
```

- **f3**
  - **@a=-4[LB]**
  - **@b=-3[LB]**
  - **@r=-2[LB]**
- **f2**
  - **@b=-5[LB]**
  - **@x=-4[LB]**
  - **@y=-2[LB]**
  - **@x1=3[LB]**
  - **@x2=4[LB]**
  - **@res=5[LB]**
- **f1**
  - **@i=-4[LB]**
  - **@r=-3[LB]**
  - **@n=-1[LB]**
  - **@r1=3[LB]**
- **test**
  - **@x=0[SB]**
  - **@y=1[SB]**
  - **@z=3[SB]**

(3 : taille de l'enregistrement d'activation)

2 Passe de placement mémoire

Nous rappelons qu'un compilateur fonctionne par passes, chacune d'elle réalisant un traitement particulier (gestion des identifiants, typage, placement mémoire, génération de code,...). Chaque passe parcourt, et potentiellement modifie, l'AST. La troisième passe est une passe de placement mémoire.

2.1 Structure de l'AST après la passe de placement mémoire

La passe de placement calcule l'adresse des variables dans la pile. Cette adresse doit être mise à jour dans les informations associées aux identificateurs.

▷ **Exercice 2** *Définir la structure de l'AST après la passe de placement mémoire.*

- **Les informations des identifiants seront mis à jour.**
- **Le reste peut être inchangé mais il s'avère que la liste des paramètres n'est plus nécessaire (ce n'est pas évident tant qu'on n'a pas vu la génération de code).**

Voici le type pour l'arbre après la passe de placement mémoire :

```
(* Expressions existantes dans notre langage *)
(* = expression de AstType *)
type expression = AstType.expression

(* instructions existantes dans notre langage *)
(* = instructions de AstType *)
type bloc = instruction list
 and instruction = AstType.instruction
```

```
(* informations associées à l'identificateur (dont son nom), liste de paramètres, corps, expression de retour *)
(* Plus besoin de la liste des paramètres mais on la garde pour les tests du placements mémoire *)
type fonction = Fonction of Tds.info_ast * Tds.info_ast list * instruction list * expression

(* Structure d'un programme dans notre langage *)
type programme = Programme of fonction list * bloc
```

2.2 Actions à réaliser lors de la passe de placement mémoire

Comme précédemment, la passe de placement est une transformation d'un arbre vers un autre, mais en fait on construit le même arbre. Les actions importantes sont la mise en jour des informations de placement dans les InfoVar.

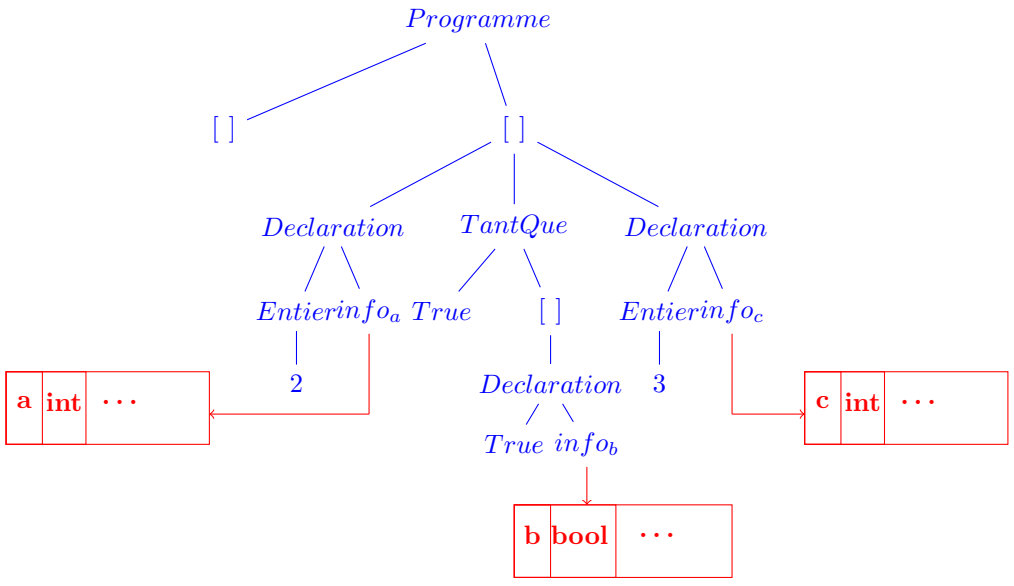
▷ **Exercice 3** *Définir les actions à réaliser lors de la passe de placement mémoire.*

On prend un exemple simple :

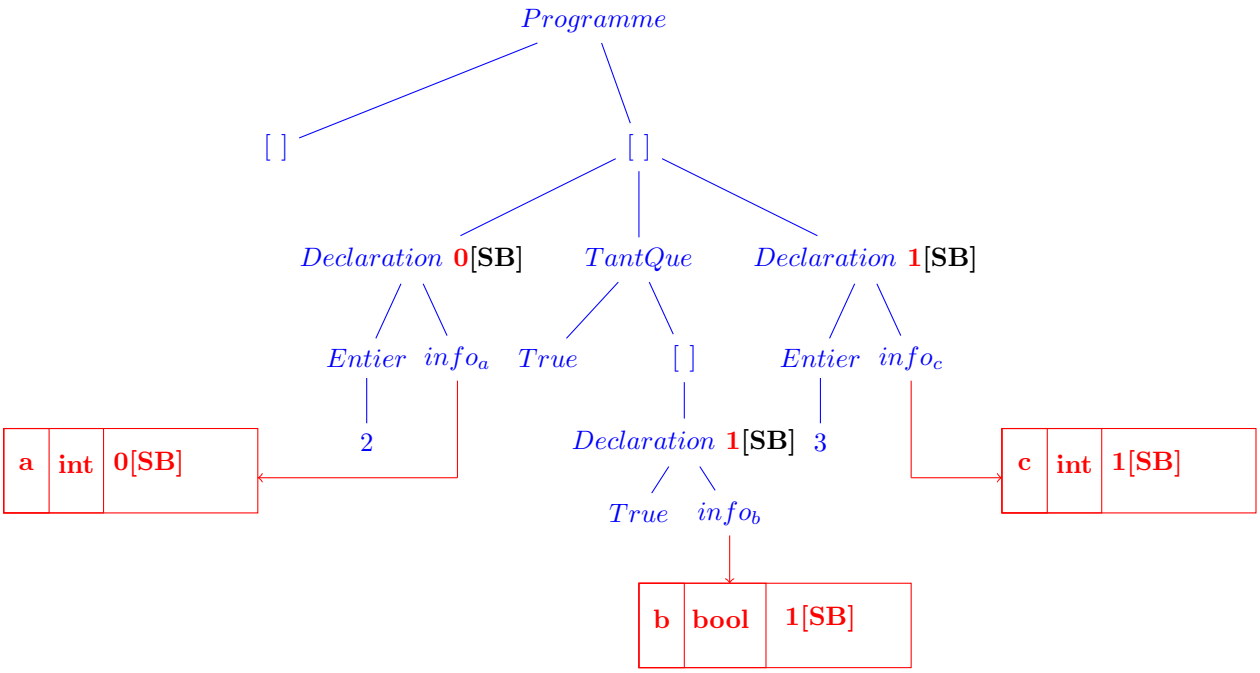
```
dep {
  int a = 2;
  while(true){
    bool b = true;
    }
  int c = 3;
}
```

On a : @a=0[SB], @b=1[SB], @c=1[SB]

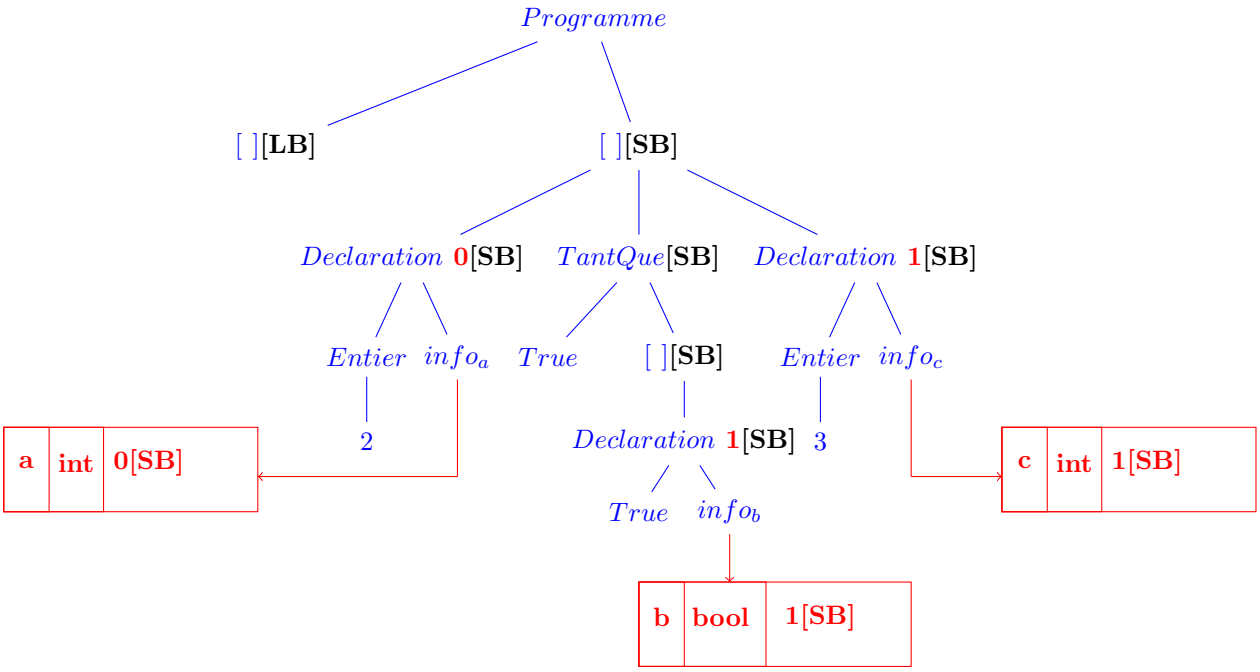
L'AstType (après la phase de typage) est le suivant :



Ce qu'on veut obtenir :

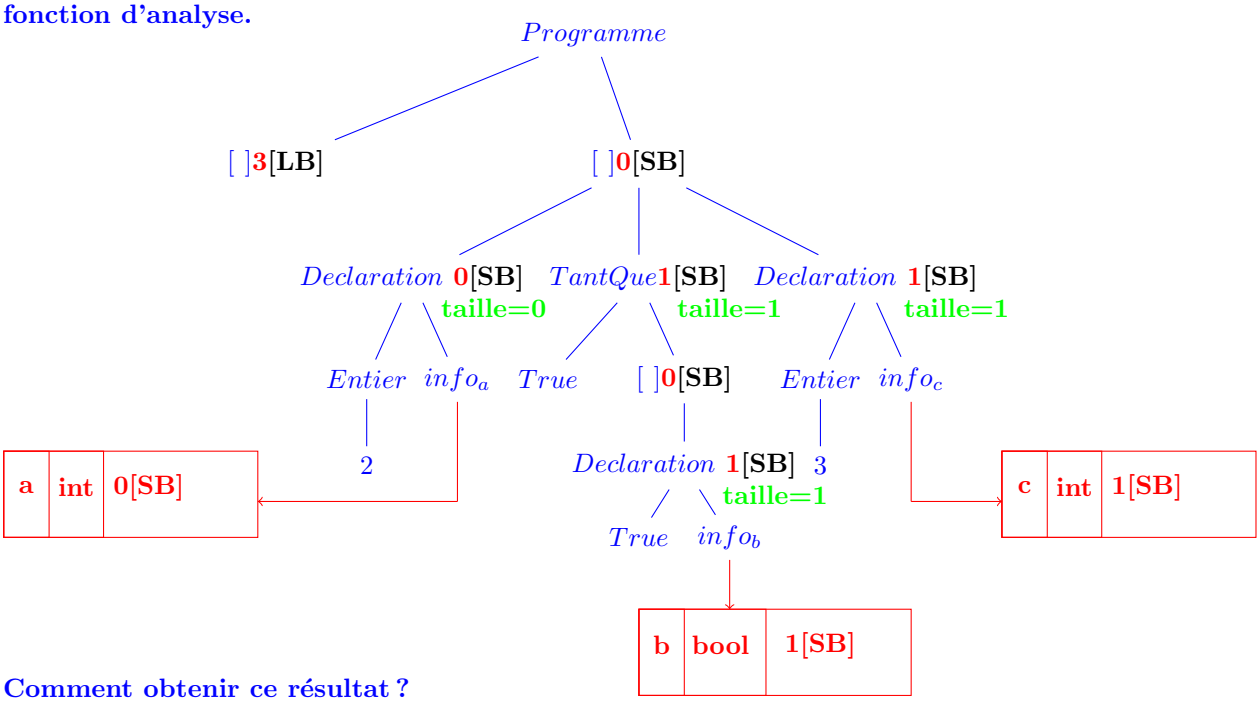


Les variables locales du programme principal et des fonctions ne sont pas placées en référence au même registre → il faut descendre le registre dans l'arbre → paramètre de la fonction d'analyse.



Pour placer les variables, il faut descendre le déplacement dans l'arbre → paramètre de la fonction d'analyse.

Pour calculer le déplacement de l'instruction suivante, on a besoin de la taille de l'instruction précédente. Cette taille remonte dans l'arbre → valeur de retour de la fonction d'analyse.



Comment obtenir ce résultat ?

- **Registre et déplacement en paramètre de la fonction d'analyse**
- **Taille qui remonte des instructions**
- **Pas besoin de descendre dans les expressions**
- **Majoritairement des effets de bord sur les informations associées aux identifiants**

Regardons d'abord l'analyse des instructions (il n'y a rien à faire pour l'analyse des expressions) :

```
let rec analyse_placement_instruction i base reg =
  match i with
  | Declaration (e, info) ->
      - (info_ast_to_info info) doit être un InfoVar (sinon erreur interne)
      - mettre à jour l'emplacement (modifier_adresse_info base reg info)
      - renvoyer la taille du type contenu dans info
  | Conditionnelle (c,t,e) ->
      analyse_placement_bloc t base reg; (* Analyse du bloc then *)
      analyse_placement_bloc e base reg; (* Analyse du bloc else *)
      0                                  (* aucun place occupée ensuite *)
  | TantQue (c,b) ->
      analyse_placement_bloc b base reg;
      0
  | _ -> 0
and analyse_placement_bloc li base reg =
    - analyser successivement chacune des instructions,
      en faisant progresser la base par addition de la taille remontée par chacune.
```

Pour les fonctions, décomposer la partie paramètres et la partie variables locales :

```
(* info_ast -> int -> int : (info, base courante) -> taille occupée *)
let analyse_placement_parametre info base =
    match info_ast_to_info info with
    | InfoVar(t,_,_) ->
        let _ = modifier_adresse_info (base - getTaille t) "LB" info in getTaille t
    | _ -> failwith ("Internal_error:_parameter_not_found")

(* info_ast list -> int *)
let rec analyse_placement_parametres lp =
(*- placer successivement les paramètres en tenant compte de la taille renvoyée
    par chaque placement.
  - ATTENTION : il faut commencer par le dernier (qui est le plus proche de LB)
  - ATTENTION : on décroît : une signature (int x, rat y, rat z) donne z en -2[LB],
    y en -4[LB], x en -5[LB] *)
let rec analyse_placement_parametres lp =
  (* inversion de la liste pour placer le dernier paramètre en sommet de pile
     puis les autres en-dessous *)
  List.fold_left (fun d p -> d + analyse_placement_parametre p (-d)) 0 (List.rev lp)

(* version compréhensible !
let rec analyse_placement_parametres lp =
    match lp with
  | [] -> 0
  | t::q ->
      let tailleq = analyse_placement_parametres q in
      let taillet = analyse_placement_parametre t (-tailleq) in
      tailleq + taillet
*)

(* AstType.fonction -> AstPlacement.fonction *)
let analyse_placement_fonction (AstType.Fonction(n,lp,li,e,info)) =
    - analyser les paramètres lp pour les placer
    - analyser le bloc pour placer les variables locales : analyse_placement_bloc li 3 "LB"
      (3 = taille enregistrement d'activation)
    - renvoyer Fonction(n,li,e,info)

(* AstType.programme -> AstPlacement.programme *)
let analyser (AstType.Programme (fonctions,prog)) =
  - analyser toutes les fonctions
  - analyser le programme principale pour placer les variables par rapport à (0,"SB")
  - retourner Programme (nfonctions, prog)
```