Traduction des langages

Arbre syntaxique abstrait, table des symboles et résolution des identifiants

Objectif :

- Définir l'AST (Abstract Syntax Tree) pour le langage RAT
- Comprendre le traitement réalisé par la passe de résolution des identifiants
- Comprendre l'intérêt de la table des symboles
- Définir la nouvelle structure d'arbre obtenue après la passe de résolution des identifiants
- Définir les actions à réaliser par la passe de résolution des identifiants

1 Le langage Rat

Cours : transparents de présentation de Rat pages 80-82

1.1 Grammaire du langage RAT

1. $PROG' \rightarrow PROG\$$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ \{IS\ return\ E\ ;\}$
4. $PROG \rightarrow\ id\ \ BLOC$
5. $BLOC \rightarrow\ \{\ IS\ \}$
6. $IS \rightarrow\ I\ \ IS$
7. $IS \rightarrow \Lambda$
8. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow\ id\ =\ E\ ;$
10. $I \rightarrow\ const\ \ id\ =\ entier\ ;$
11. $I \rightarrow\ print\ \ E;$
12. $I \rightarrow\ if\ \ E\ \ BLOC\ \ else\ \ BLOC$
13. $I \rightarrow\ while\ \ E\ \ BLOC$
14. $DP \rightarrow \Lambda$
15. $DP \rightarrow TYPE\ id\ DP$
16. $TYPE \rightarrow\ bool$
17. $TYPE \rightarrow\ int$
18. $TYPE \rightarrow\ rat$
19. $E \rightarrow call\ id\ (\ CP\ )$
20. $CP \rightarrow \Lambda$
21. $CP \rightarrow E\ CP$
22. $E \rightarrow\ [\ E\ /\ E\ ]$
23. $E \rightarrow\ num\ \ E$
24. $E \rightarrow\ denom\ \ E$
25. $E \rightarrow\ id$
26. $E \rightarrow\ true$
27. $E \rightarrow\ false$
28. $E \rightarrow\ entier$
29. $E \rightarrow\ (\ E\ +\ \ E\ )$
30. $E \rightarrow\ (\ E\ *\ \ E\ )$
31. $E \rightarrow\ (\ E\ =\ \ E\ )$
32. $E \rightarrow\ (\ E\ <\ \ E\ )$
33. $E \rightarrow\ (\ E\ \ )$

1.2 Exemple

```
bool less ( rat a rat b ){
  return ((num a * denom b )<( num b * denom a ));
}

prog{
```

```
  rat a = [3/4];
  rat b = [4/5];
  const n = 5;
  int i = 0;
  while(i<n){
    a=(a+a);
    b=(b*b);
    i=(i+1);
  }
  if(call less(a b)){print a;} else {print b;}
}
```

2 Structure de l'AST pour RAT

Cours : partie V sur l'AST.

Les transparents 83 - 85 montrent, pour un exemple de code RAT, l'arbre de dérivation et l'AST retenu.

▷ **Exercice 1** *Définir la structure de l'AST pour RAT.*

Voici le type utilisé dans les TP :

```
module AstSyntax =
struct

(* Opérateurs binaires de Rat *)
type binaire = Plus | Mult | Equ | Inf

(* Expressions de Rat *)
type expression =
  (* Appel de fonction représenté par le nom de la fonction et la liste des paramètres réels *)
  | AppelFonction of string * expression list
  (* Rationnel représenté par le numérateur et le dénominateur *)
  | Rationnel of expression * expression
  (* Accès au numérateur d'un rationnel *)
  | Numerateur of expression
  (* Accès au dénominateur d'un rationnel *)
  | Denominateur of expression
  (* Accès à un identifiant représenté par son nom *)
  | Ident of string
  (* Booléen vrai *)
  | True
  (* Booléen faux *)
  | False
  (* Entier *)
  | Entier of int
  (* Opération binaire représentée par l'opérateur, l'opérande gauche et l'opérande droite *)
  | Binaire of binaire * expression * expression

(* Instructions de Rat *)
type bloc = instruction list
and instruction =
```

```
  (* Déclaration de variable représentée par son type, son nom et l'expression d' initialisation *)
  | Declaration of typ * string * expression
  (* Affectation d'une variable représentée par son nom et la nouvelle valeur affectée *)
  | Affectation of string * expression
  (* Déclaration d'une constante représentée par son nom et sa valeur (entier) *)
  | Constante of string * int
  (* Affichage d'une expression *)
  | Affichage of expression
  (* Conditionnelle représentée par la condition, le bloc then et le bloc else *)
  | Conditionnelle of expression * bloc * bloc
  (*Boucle TantQue représentée par la conditin d'arrêt de la boucle et le bloc d'instructions *)
  | TantQue of expression * bloc

(* Structure des fonctions de Rat *)
(* type de retour - nom - liste des paramètres (association type et nom) - corps de la fonction - expression de retour *)
type fonction = Fonction of typ * string * (typ * string) list * instruction list * expression

(* Structure d'un programme Rat *)
(* liste de fonction - programme principal *)
type programme = Programme of fonction list * bloc

end
```

3 Passe de résolution des identifiants

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 première passe est une passe de résolution des identifiants. C'est elle qui vérifie la bonne utilisation des identifiants. En particulier, dans le cas du langage RAT, c'est elle qui vérifie que :

- déclaration : vérifier l'absence de double déclaration dans le même bloc ;
- utilisation d'un symbole : vérifier l'existence de ce symbole dans ce bloc ou un bloc englobant ;
- utilisation d'un symbole : vérifier que sa catégorie (variable, constante, fonction) est correcte.

Cours : audio du transparent 86

Pour réaliser cette passe, nous avons besoin d'une table des symboles : structure de données associant aux identifiants leurs informations.

3.1 Table des symboles

Cours : transparents 86 à 91 (table des symboles).

L'interface du module Tds utilisé en TP est la suivante :

```
(* Définition du type des informations associées aux identifiants *)
type info =
  (* Information associée à une constante : son nom (non indispensable mais aide au test et debbugage) et sa valeur *)
  | InfoConst of string * int
  (* Information associée à une variable : son nom (non indispensable mais aide au test et debbugage),
  son type, et son adresse ie son déplacement (int) par rapport à un registre (string) *)
  | InfoVar of string * typ * int * string
```

```
  (* Information associée à une fonction : son nom (utile pour l'appel), son type de retour
     et la liste des types des paramètres *)
  | InfoFun of string * typ * typ list

(* Table des symboles *)
type tds

(* Données stockées dans la tds et dans les AST : pointeur sur une information *)
type info_ast

(* Création d'une table des symboles à la racine *)
val creerTDSMere : unit -> tds

(* Création d'une table des symboles fille *)
(* Le paramètre est la table mère *)
val creerTDSFille : tds -> tds

(* Ajoute une information dans la table des symboles locale *)
(* tds : la tds courante *)
(* string : le nom de l'identificateur *)
(* info : l'information à associer à l'identificateur *)
(* Si l'identificateur est déjà présent dans TDS, l'information est écrasée *)
(* retour : unit *)
val ajouter : tds -> string -> info_ast -> unit

(* Recherche les informations d'un identificateur dans la tds locale *)
(* Ne cherche que dans la tds de plus bas niveau *)
val chercherLocalement : tds -> string -> info_ast option

(* Recherche les informations d'un identificateur dans la tds globale *)
(* Si l'identificateur n'est pas présent dans la tds de plus bas niveau la recherche est effectuée *)
(* dans sa table mère et ainsi de suite jusqu'à trouver (ou pas) l'identificateur *)
val chercherGlobalement : tds -> string -> info_ast option

(* Affiche la tds locale *)
val afficher_locale : tds -> unit

(* Affiche la tds locale et récursivement *)
val afficher_globale : tds -> unit

(* Créer une information à associer à l'AST àpartir d'une info *)
val info_to_info_ast : info -> info_ast

(* Récupère l'information associée à un noeud *)
val info_ast_to_info : info_ast -> info

(* Modifie le type si c'est une InfoVar, ne fait rien sinon *)
val modifier_type_info : typ -> info_ast -> unit

(* Modifie les types de retour et des paramètres si c'est une InfoFun, ne fait rien sinon *)
val modifier_type_fonction_info : typ -> typ list -> info_ast -> unit

(* Modifie l'emplacement (dépl, registre) si c'est une InfoVar, ne fait rien sinon *)
```

```
val modifier_adresse_info : int -> string -> info_ast -> unit
```

- **`info_ast` est un pointeur sur une `info`. On utilisera à bon escient `info_ast_to_info` pour obtenir la valeur (déréférencement du pointeur) et `info_to_info_ast` exclusivement pour créer le pointeur pour l'insérer dans la table des symboles.**
- **Cette partie de l'UE fait exceptionnellement une entorse à la programmation sans effet de bord : on autorise les effets de bord sur la tds et les infos. En effet, les passes suivantes (typage, placement mémoire) vont mettre à jour les informations associées aux symboles. Il est plus simple de modifier l'information associée que de reconstruire une nouvelle tds.**

3.2 Structure de l'AST post passe de résolution des identifiants

La passe de résolution des identifiants réalise des vérifications mais prépare également les passes suivantes. Elles auront besoin d'avoir accès aux informations des identifiants, il faut donc modifier l'AST de façon à rendre ces informations accessibles.

▷ **Exercice 2** *Définir la structure de l'AST post passe de résolution des identifiants.*

- **On associe les informations des identifiants à tous les nœuds où il y a des identifiants.**
- **Puisque les identifiants ont été résolus, il n'est plus nécessaire de les garder dans l'arbre.**
- **Comme les valeurs des constantes sont stockées dans l'info associée à la constante, il est inutile de garder leur déclaration dans l'AST.**

Voici un type possible pour l'arbre après la passe de résolution des identifiants :

```
module AstTds =
struct
 (* Expressions existantes dans notre langage *)
  (* ~ expression de l'AST + informations associées aux identificateurs *)
  type expression =
    | AppelFonction of Tds.info_ast * expression list
    | Rationnel of expression * expression
    | Numerateur of expression
    | Denominateur of expression
    | Ident of Tds.info_ast
    | True
    | False
    | Entier of int
    | Binaire of AstSyntax.binaire * expression * expression

  (* instructions existantes dans notre langage *)
  (* = instruction de l'AST + infos associées aux identificateurs + suppression de nœuds *)
  type bloc = instruction list
  and instruction =
    | Declaration of typ * expression * Tds.info_ast
    | Affectation of expression * Tds.info_ast
    | Affichage of expression
    | Conditionnelle of expression * bloc * bloc
    | TantQue of expression * bloc
```

```
  | Empty (* les nœuds ayant disparus: Const *)


  (* Structure des fonctions dans notre langage *)
  (* type de retour, informations associées à l'identificateur (dont son nom), liste des paramètres,
     corps, expression de retour *)
  type fonction = Fonction of typ * Tds.info_ast * (typ * Tds.info_ast ) list *
                                instruction list * expression

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

On ne peut pas se contenter d'associer l'information de l'identifiant, car on a besoin d'avoir des modifications par effet de bord ⇒ c'est au niveau de la déclaration que les informations sont connues, chaque passe va mettre à jour petit à petit ces informations ; mais c'est au niveau des utilisations qu'on en a besoin. On a donc un pointeur sur les informations.

3.3 Actions à réaliser lors de la passe de résolution des identifiants

La passe de résolution des identifiants est donc une transformation d'un arbre vers un autre.

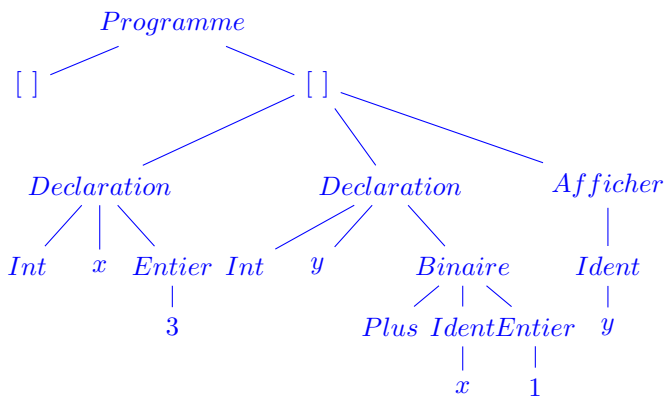
▷ **Exercice 3** *Définir les actions à réaliser lors de la passe de résolution des identifiants.*

Considérons l'exemple simple du cours et voyons ce qu'on veut obtenir.

1. **Code :**

```
main{
 int x = 3;
 int y = x+1;
 print y;
}
```

2. **AstSyntax**



3. **Actions à réaliser**

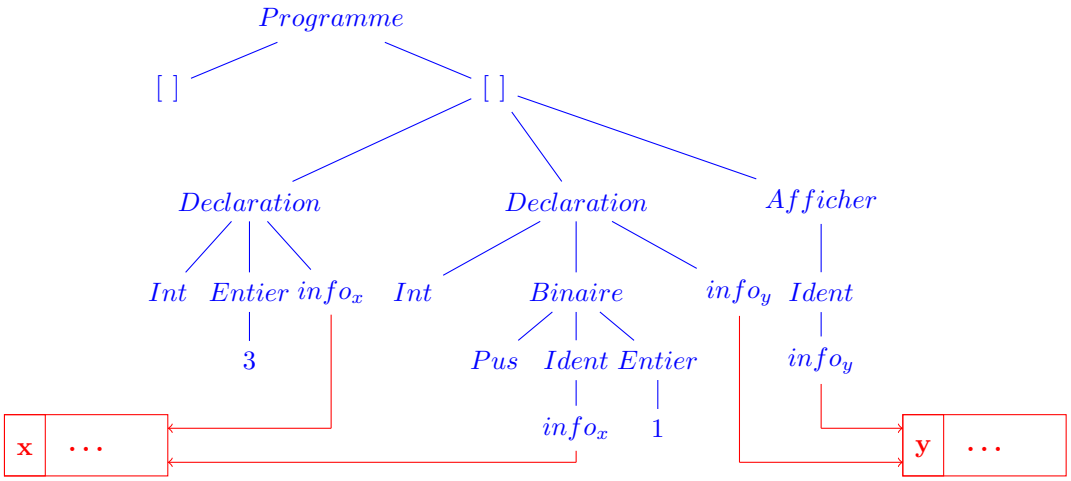
(a) **Instruction 1 : vérifier que x n'a pas déjà été déclaré**

(b) **Instruction 2 : vérifier que y n'a pas déjà été déclaré**

(c) **Instruction 2 : vérifier que x a déjà été déclaré comme variable ou constante**

(d) **Instruction 3 : vérifier que y a déjà été déclaré comme variable ou constante**

4. **AstTds**



$info_x$ et $info_y$ sont des pointeurs, donc quand l'information sera modifiée au niveau de la déclaration, cette modification sera visible au niveau de l'utilisation.

Comment obtenir ce résultat ?

- **On propage une tds dans l'arbre. Elle sera modifiée par effet de bord (pas besoin de la faire remonter) ;**
- **Au niveau des déclarations, on crée une information qu'on ajoute à la tds ;**
- **Au niveau des utilisations, on va chercher l'information dans la tds, on contrôle la bonne utilisation, et on l'attache au nœud ;**
- **Pour faire circuler la tds, elle est passée en paramètre de la fonction d'analyse.**

Le plus simple est de commencer par l'analyse des instructions :

```
(* Ast.instruction -> tds -> Asttds.instruction *)
let rec analyse_tds_instruction tds i =
  match i with
  | AstSyntax.Declaration (t, id, e) ->
      On cherche localement id dans la tds
      Si il est présent
      Alors Erreur "double_déclaration"
      Sinon
        - on analyse l'expression et on récupère le nouvel arbre associé (ne)
        - on ajoute l' identifiant à la tds (InfoVar (id, Undefined, 0, ""))
        - on renvoie le nouvel arbre : Declaration (t, ne, info)
  | AstSyntax.Affectation (id,e) ->
      On cherche globalement id dans la tds
```

```
      Si il est présent
      Alors
         Si c'est un InfoVar
         Alors
            - on analyse l'expression et on récupère le nouvel arbre associé (ne)
            - on renvoie le nouvel arbre : Affectation (ne, info)
         Sinon Erreur "mauvaise utilisation d'un identifiant (modification de constante ou fonction)"
      Sinon Erreur " identifiant non déclaré"
  | AstSyntax.Constante (id,v) ->
      On cherche localement id dans la tds
      Si il est présent
      Alors Erreur "double déclaration"
      Sinon
         - on ajoute l' identifiant à la tds (InfoConst (id,v))
         - on renvoie le nouvel arbre : Empty
  | AstSyntax.Affichage e ->
      - on analyse l'expression et on récupère le nouvel arbre associé (ne)
      - on renvoie le nouvel arbre : Affichage (ne)
  | AstSyntax.Conditionnelle (c,t,e) ->
      (* Analyse de la condition *)
      let nc = analyse_tds_expression tds c in
      (* Analyse du bloc then *)
      let tast = analyse_tds_bloc tds t in
      (* Analyse du bloc else *)
      let east = analyse_tds_bloc tds e in
      (* les variables sont déclarées localement aux blocs *)
      Conditionnelle (nc, tast, east)
  | AstSyntax.TantQue (c,b) ->
      (* Analyse de la condition *)
      let nc = analyse_tds_expression tds c in
      (* Analyse du bloc *)
      let bast = analyse_tds_bloc tds b in
      TantQue (nc, bast)
```

On peut alors passer à l'analyse du bloc :

```
(* Ast.bloc -> Asttds.bloc *)
and analyse_tds_bloc tds li =
  - création de la tds locale au bloc
  - analyse de chacune des instructions du bloc avec la tds locale au bloc
     => map
```

Puis on passe à l'analyse des expressions :

```
let rec analyse_tds_expression tds e =
  match e with
  | AstSyntax.AppelFonction (id,le) ->
      On cherche globalement id dans la tds
      S'il est présent
      Alors
         Si c'est un InfoFun
         Alors
            - on analyse la liste des paramètres (map) et on récupère la nouvelle forêt associée (ne)
            - on renvoie le nouvel arbre : AppelFonction (info, ne)
```

```
        Sinon Erreur "mauvaise utilisation d'un identifiant (appel d'autre chose qu'une fonction)"
      Sinon Erreur " identifiant non déclaré"
  | AstSyntax.Ident n ->
      On cherche globalement id dans la tds
      S'il est présent
      Alors
        Si c'est un InfoVar alors on renvoie le nouveau noeud : Ident info
        Si c'est un InfoConst(_,v) alors on remplace la constante : Entier v
        Sinon Erreur "mauvaise utilisation d'un identifiant ( utilisation d'une fonction sans call)"
      Sinon Erreur " identifiant non déclaré"
  | AstSyntax.Rationnel (e1,e2) ->
      Rationnel (analyse_tds_expression tds e1 , analyse_tds_expression tds e2)
  | ...
```

La suite est sur le même schéma car il n'y a pas d'identifiant :

- **on propage la tds et la vérification dans tout l'arbre**
- **on reconstruit l'arbre avec ce qui remonte de l'analyse**

On peut alors faire l'analyse du programme :

```
let analyser (AstSyntax.Programme (fonctions,prog)) =
  - création de la TDS initiale : vide
  - analyse des fonctions et récupération du nouvel arbre associé aux fonctions (nf)
  - analyse du bloc principal et récupération du nouvel arbre associé au bloc principal (nb)
  - renvoi du nouvel arbre : Programme (nf, nb)
```

Enfin, il reste à faire l'analyse des fonctions : réfléchir et coder.

Traduction des langages

Typage

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

1 Rappel : le typage du langage Rat

1.1 Grammaire du langage RAT

1. $PROG' \rightarrow PROG\$$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ \{IS\ return\ E\ ;\}$
4. $PROG \rightarrow\ id\ \ BLOC$
5. $BLOC \rightarrow\ \{\ IS\ \}$
6. $IS \rightarrow\ I\ \ IS$
7. $IS \rightarrow \Lambda$
8. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow\ id\ =\ E\ ;$
10. $I \rightarrow\ const\ \ id\ =\ entier\ ;$
11. $I \rightarrow\ print\ \ E;$
12. $I \rightarrow\ if\ \ E\ \ BLOC\ \ else\ \ BLOC$
13. $I \rightarrow\ while\ \ E\ \ BLOC$
14. $DP \rightarrow \Lambda$
15. $DP \rightarrow TYPE\ id\ DP$
16. $TYPE \rightarrow\ bool$
17. $TYPE \rightarrow\ int$
18. $TYPE \rightarrow\ rat$
19. $E \rightarrow call\ id\ (\ CP\ )$
20. $CP \rightarrow \Lambda$
21. $CP \rightarrow E\ CP$
22. $E \rightarrow\ [\ E\ /\ E\ ]$
23. $E \rightarrow\ num\ \ E$
24. $E \rightarrow\ denom\ \ E$
25. $E \rightarrow\ id$
26. $E \rightarrow\ true$
27. $E \rightarrow\ false$
28. $E \rightarrow\ entier$
29. $E \rightarrow\ (\ E\ +\ \ E\ )$
30. $E \rightarrow\ (\ E\ *\ \ E\ )$
31. $E \rightarrow\ (\ E\ =\ \ E\ )$
32. $E \rightarrow\ (\ E\ <\ \ E\ )$
33. $E \rightarrow\ (\ E\ )$

1.2 Jugements de typage du langage RAT

Axiomes

- $(x, \tau) :: \sigma \vdash x : \tau$
- $\dfrac{\sigma \vdash x : \tau}{(y, \tau') :: \sigma \vdash x : \tau}$
- $\sigma \vdash true : bool$
- $\sigma \vdash false : bool$
- $\sigma \vdash entier : int$

Expression

— $\dfrac{\sigma \vdash E_1 : int \qquad \sigma \vdash E_2 : int}{\sigma \vdash [\ E_1\ /\ E_2\ ] : rat}$

— $\dfrac{\sigma \vdash E : rat}{\sigma \vdash num\ E : int}$

— $\dfrac{\sigma \vdash E : rat}{\sigma \vdash denom\ E : int}$

— $\dfrac{\sigma \vdash E : \tau}{\sigma \vdash (E) : \tau}$

— $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 + E_2) : int}$

— $\dfrac{\sigma \vdash E_1 : rat \quad \sigma \vdash E_2 : rat}{\sigma \vdash (E_1 + E_2) : rat}$

— $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 * E_2) : int}$

— $\dfrac{\sigma \vdash E_1 : rat \quad \sigma \vdash E_2 : rat}{\sigma \vdash (E_1 * E_2) : rat}$

— $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 = E_2) : bool}$

— $\dfrac{\sigma \vdash E_1 : bool \quad \sigma \vdash E_2 : bool}{\sigma \vdash (E_1 = E_2) : bool}$

— $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 < E_2) : bool}$

— On se limitera à ces signatures.

Structures de contrôle

— $\dfrac{\sigma \vdash E : bool \qquad \sigma \vdash BLOC_1 : void \qquad \sigma \vdash BLOC_2 : void}{\sigma \vdash if\ E\ BLOC_1\ else\ BLOC_2 : void, []}$

— $\dfrac{\sigma \vdash E : bool \qquad \sigma \vdash BLOC : void}{\sigma \vdash while\ E\ BLOC : void, []}$

Déclaration / affectation

— $\dfrac{\sigma \vdash TYPE : \tau \qquad \sigma \vdash E : \tau}{\sigma \vdash TYPE\ id = E : void, [id, \tau]}$

— $\dfrac{\sigma \vdash id : \tau \qquad \sigma \vdash E : \tau}{\sigma \vdash id = E : void, []}$

Autres instructions

— $\dfrac{}{\sigma \vdash const\ \ id\ =\ entier : void, [id, int]}$

— $\dfrac{\sigma \vdash E : \tau}{\sigma \vdash print\ \ E : void, []}$

Déclaration de fonction

— $\dfrac{A \quad B \quad C \quad D}{\sigma \vdash TYPE\ id\ (DP)\ \{IS\ return\ E;\} : void, [id, \tau_2 \to \tau_1]}$

— A : $\sigma \vdash TYPE : \tau_1$
— B : $\sigma \vdash DP : \tau_2, \sigma_p \qquad (DP = TYPE_1\ id_1 \ldots TYPE_n\ id_n)$
— C : $(id, \tau_2 \to \tau_1)@\sigma_p@\sigma \vdash IS : void, \sigma_l$
— D : $(id, \tau_2 \to \tau_1)@\sigma_l@\sigma_p@\sigma \vdash E : \tau_1$

— $\dfrac{\sigma \vdash TYPE_1 : \tau_1 \quad \ldots \quad TYPE_n : \tau_n}{\sigma \vdash TYPE_1\ id_1\ \ldots\ TYPE_n\ id_n : \tau_1 \times \cdots \times \tau_n, [(id_1, \tau_1); \ldots; (id_n, \tau_n)]}$

Appel de fonction

— $\dfrac{\sigma \vdash id : \tau_1 \to \tau_2 \qquad E_1 \ \ldots \ E_n : \tau_1}{\sigma \vdash call\ id\ (E_1 \ \ldots \ E_n) : \tau_2}$

— $\dfrac{\sigma \vdash E_1 : \tau_1 \quad \ldots \quad E_n : \tau_n}{\sigma \vdash E_1 \ \ldots \ E_n : \tau_1 \times \cdots \times \tau_n}$

Suite d'instructions

— $\dfrac{\sigma \vdash IS : void, \sigma'}{\sigma \vdash \{IS\} : void}$

— $\dfrac{\sigma \vdash I : void, \sigma' \qquad \sigma' @ \sigma \vdash IS : void, \sigma''}{\sigma \vdash I\ IS : void, \sigma'' @ \sigma'}$

— $\sigma \vdash \quad : void$

Le programme

— $\dfrac{\sigma \vdash FUN : void, \sigma' \qquad \sigma' @ \sigma \vdash PROG : void, \sigma''}{\sigma \vdash FUN\ PROG : void, \sigma'' @ \sigma'}$

— $\dfrac{\sigma \vdash BLOC : void}{\sigma \vdash id\ BLOC : void}$

2 Passe de typage

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 seconde passe est une passe de typage. C'est elle qui vérifie la conformité des types déclarés et associe aux identifiants leurs informations de type.

2.1 Structure de l'AST après la passe de typage

La passe de typage réalise des vérifications de type qui nécessitent une mise à jour des informations de type de identificateurs. Elle prépare également les passes suivantes, par exemple en choisissant la "version" de l'opérateur à utiliser en cas de surcharge des opérateurs.

▷ **Exercice 1** *Définir la structure de l'AST après la passe de typage.*

- **Les informations des identifiants seront mises à jour avec les informations de type.**
- **Les informations de types n'ont plus besoin d'être conservées dans les nœuds de l'arbre.**
- **Les opérateurs surchargés sont "résolus" et remplacés par des opérateurs non surchargés.**

Voici un type possible pour l'arbre après la passe de typage :

```
(* Opérateurs binaires existants dans Rat - résolution de la surcharge *)
type binaire = PlusInt | PlusRat | MultInt | MultRat | EquInt | EquBool | Inf
```

```
(* Expressions existantes dans Rat *)
(* = expression de AstTds *)
type expression =
  | AppelFonction of Tds.info_ast * expression list
  | Rationnel of expression * expression
  | Numerateur of expression
  | Denominateur of expression
  | Ident of Tds.info_ast
  | True
  | False
  | Entier of int
  | Binaire of binaire * expression * expression

(* instructions existantes Rat *)
(* = instruction de AstTds + informations associées aux identificateurs, mises à jour *)
(* + résolution de la surcharge de l'affichage *)
type bloc = instruction list
 and instruction =
  | Declaration of expression * Tds.info_ast
  | Affectation of expression * Tds.info_ast
  | AffichageInt of expression
  | AffichageRat of expression
  | AffichageBool of expression
  | Conditionnelle of expression * bloc * bloc
  | TantQue of expression * bloc
  | Empty (* les nœuds ayant disparus: Const *)

(* informations associées à l'identificateur (dont son nom), liste des paramètres, corps, expression de retour *)
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 typage

La passe de typage est donc une transformation d'un arbre vers un autre.

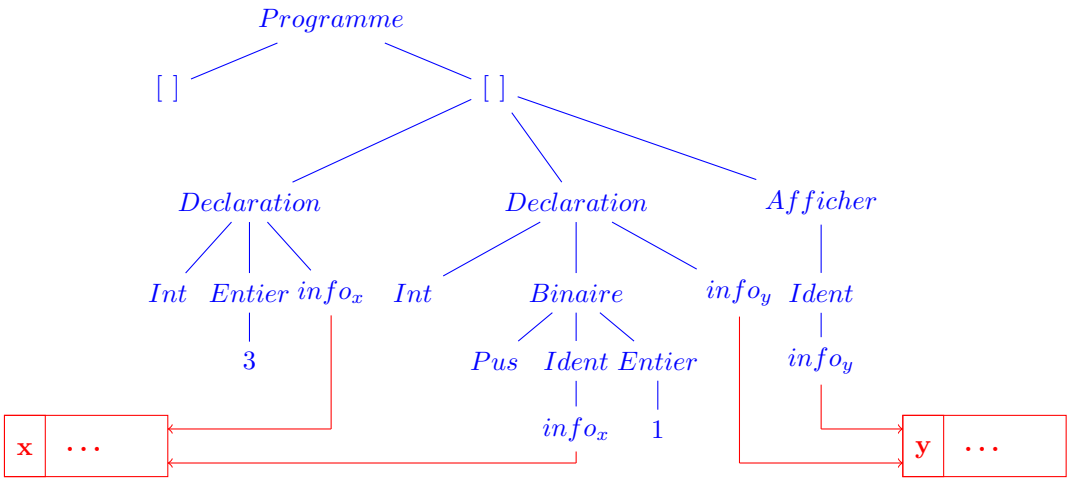
▷ **Exercice 2** *Définir les actions à réaliser lors de la passe de typage.*

On reprend l'exemple :

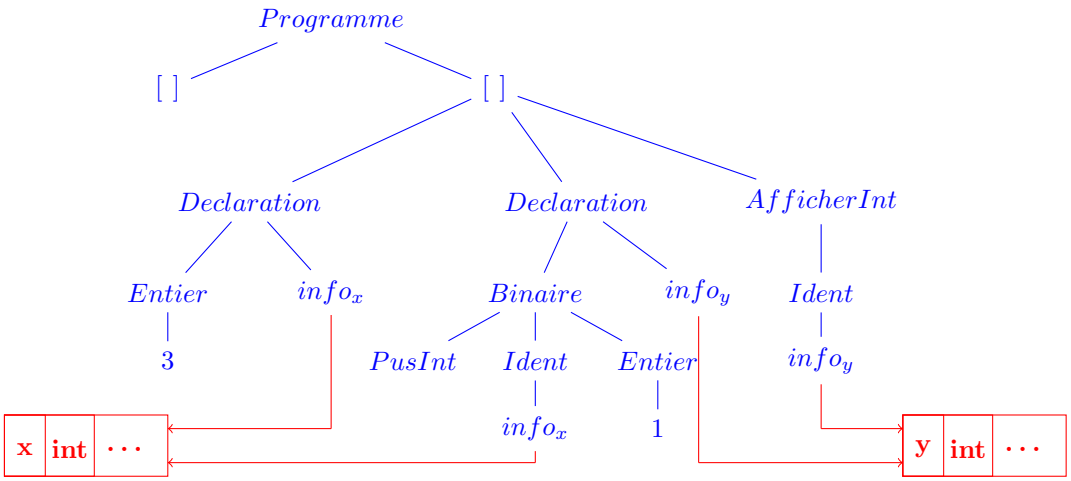
1. **Code :**

```
main{
 int x = 3;
 int y = x+1;
 print y;
}
```

2. **AstTds**



3. **Actions à réaliser :**
   (a) **Mettre à jour $info_x$ avec le type de x**
   (b) **Vérifier que + est utilisé avec une bonne signature**
   (c) **Identifier la version de + à utiliser (surcharge)**
   (d) **Vérifier que le type de retour de + est compatible avec le type déclaré de y**
   (e) **Mettre à jour $info_y$ avec le type de y**
   (f) **Identifier la version d'affichage à utiliser (surcharge)**
4. **AstType**



Comment obtenir ce résultat ? Les informations de type sont connues au niveau des feuilles, il faut donc que les informations de types remontent dans l'arbre ⇒ valeur de retour des fonctions d'analyse. Nous n'en avons besoin que sur les expressions, car les instructions sont typées en unit / void.

Commençons par l'analyse des instructions puis des expressions. L'analyse du programme principal et l'analyse des fonctions ne sont pas détaillées et sont à définir.

```
let rec analyse_type_instruction i =
  match i with
  | AstTds.Declaration (t, e, info) ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - vérifier que te est compatible avec t [ sinon TypeInattendu ]
        - mettre à jour le type dans info (modifier_type_info t info);
        - retourner Declaration (ne, info)
  | AstTds.Affectation (e, info) ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - vérifier que te est compatible avec le type dans l'info
        - retourner Affectation (ne, info)
    (* Affichage *)
    | AstTds.Affichage e ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - selon te, construire le nouvel arbre en résolvant la surcharge
    (*Conditionnelle *)
    | AstTds.Conditionnelle (c,t,e) ->
        - analyser c pour obtenir son type (tc) et le nouvel arbre (nc)
        - vérifier que tc est Bool [ sinon TypeInattendu ]
        - analyser les blocs t et e
        - retourner Conditionnelle(nc,nt,ne)
    (* Tant Que *)
    | AstTds.TantQue (c,b) ->
      - analyser c pour obtenir son type (tc) et le nouvel arbre (nc)
      - vérifier que tc est Bool [ sinon TypeInattendu ]
      - analyser le bloc b
      - retourner TantQue(nc,nb)
    | AstTds.Empty -> Empty

and analyse_type_bloc b = map analyse_type_instruction b
```

Puis on passe aux expressions :

```
(* Asttds.ast -> (ast,typ) *)
let rec analyse_type_expression e =
  match e with
  | AstTds.AppelFonction (info,le) ->                    (* Appel de fonction *)
    - ( info_ast_to_info info) doit être un InfoFun(nom,typeRet,typeParams)
      [ sinon erreur interne, ça doit être garanti par la passe précédente ]
    - analyser chacune des expressions dans le, pour obtenir nle (nouvelle
      liste des arguments) et ltype ( liste des types de ces expressions)
    - vérifier que ltype est compatible avec typeParams
     [ sinon exception TypesParametresInattendus ]
    - retourner le couple (AppelFonction (info,nle), typeRet)
  | AstTds.Rationnel (e1,e2) ->                          (* Définition d'un rationnel *)
    - analyser e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
    - analyser e2 pour obtenir ne2 (nouvel arbre) et t2 (type calculé)
    - vérifier que t1 et t2 sont Int (sinon TypeInattendu)
    - retourner (Rationnel(ne1,ne2), Rat)
  | AstTds.Numerateur e1 ->                              (* Accès au numérateur *)
    - analyser l'expression e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
    - vérifier que t1 est Rat (sinon TypeInattendu)
```

```
  - retourner (Numerateur ne1, Int)
| AstTds.Denominateur e1 ->                          (* Accès au dénominateur *)
  - analyser l'expression e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
  - vérifier que t1 est Rat (sinon TypeInattendu)
  - retourner (Denominateur ne1, Int)
 | AstTds.Ident info ->                              (* Accès à un identifiant *)
  begin
    match info_ast_to_info info  with
    | InfoVar (_,t,_,_) -> (Ident info, t)
    | InfoConst (_,_) -> (Ident info, Int)
    | _ -> failwith ("Internal_error:_symbol_not_found")
  end
| AstTds.Binaire (b,e1,e2) ->                        (* Opération binaire *)
  - analyser e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
  - analyser e2 pour obtenir ne2 (nouvel arbre) et t2 (type calculé)
  - identifier les cas corrects et retourner le couple (arbre,type) correspondant:
    - Int, Plus, Int -> (Binaire (PlusInt,ne1,ne2), Int)
    - Rat, Plus, Rat -> (Binaire (PlusRat,ne1,ne2), Rat)
    - Int, Equ, Int  -> (Binaire (EquInt,ne1,ne2), Bool)
    - Bool, Equ, Bool -> (Binaire (EquBool,ne1,ne2), Bool)
    - etc
    - sinon TypeBinaireInattendu
| AstTds.True -> (True, Bool)
| AstTds.False -> (False, Bool)
| AstTds.Entier i -> (Entier i, Int)
```

Il reste à définir l'analyse du programme principal et l'analyse des fonctions.

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.

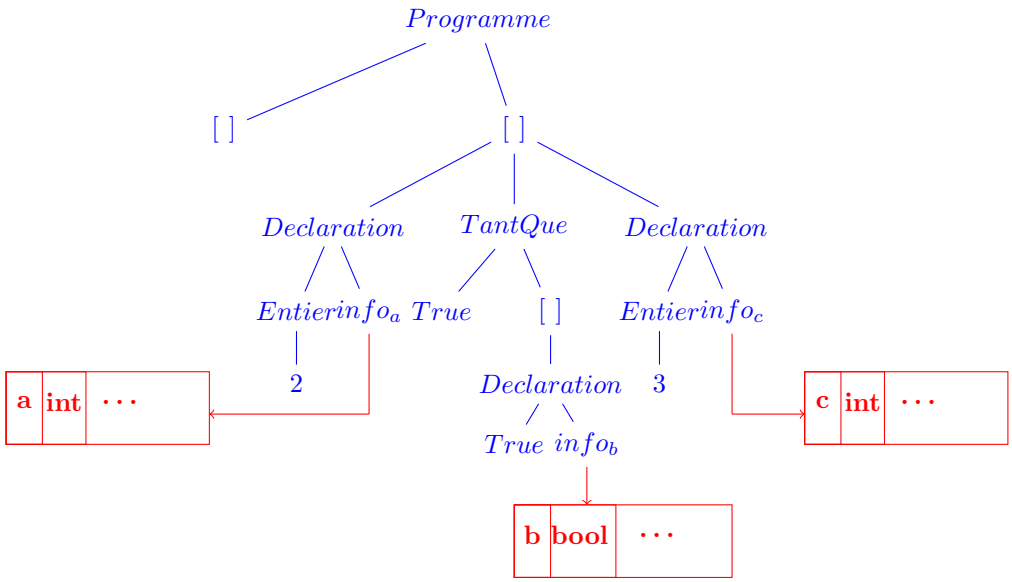
▷ **Exercice 3** *Définir les actions à réaliser lors de la passe de placement mémoire.*

On prend un exemple simple :

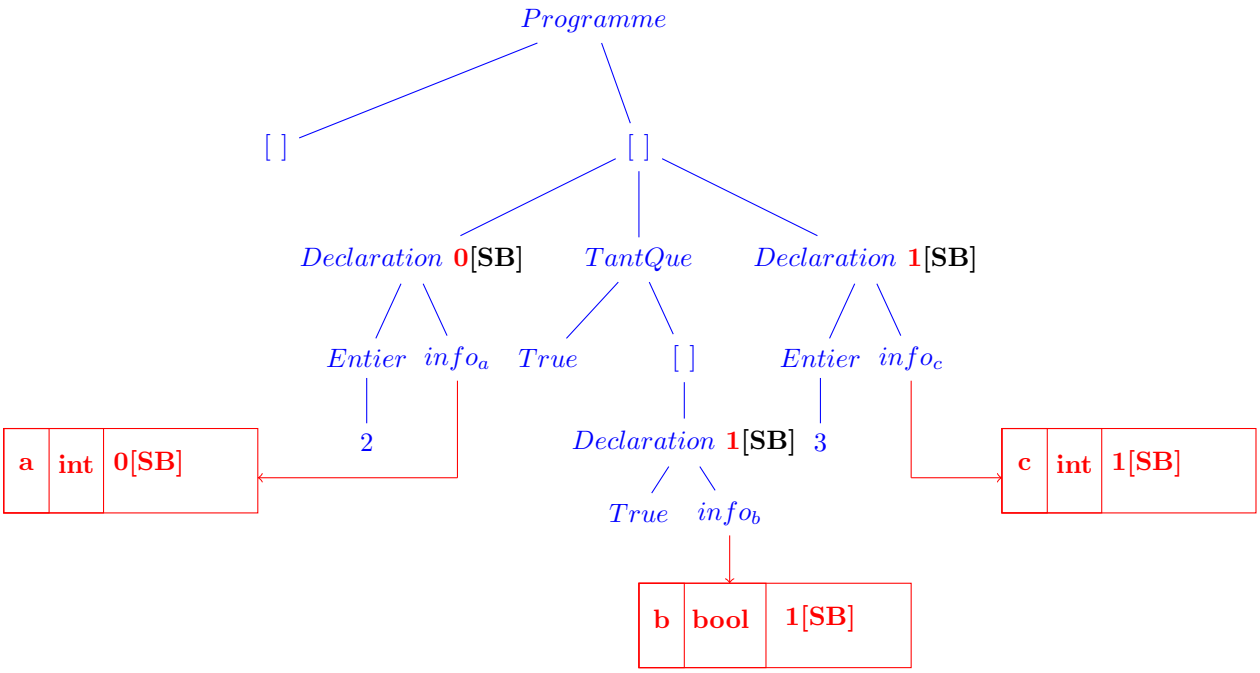
```
dep {
  int a = 2;
  while(true){
    bool b = true;
    }
  int c = 3;
}
```

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

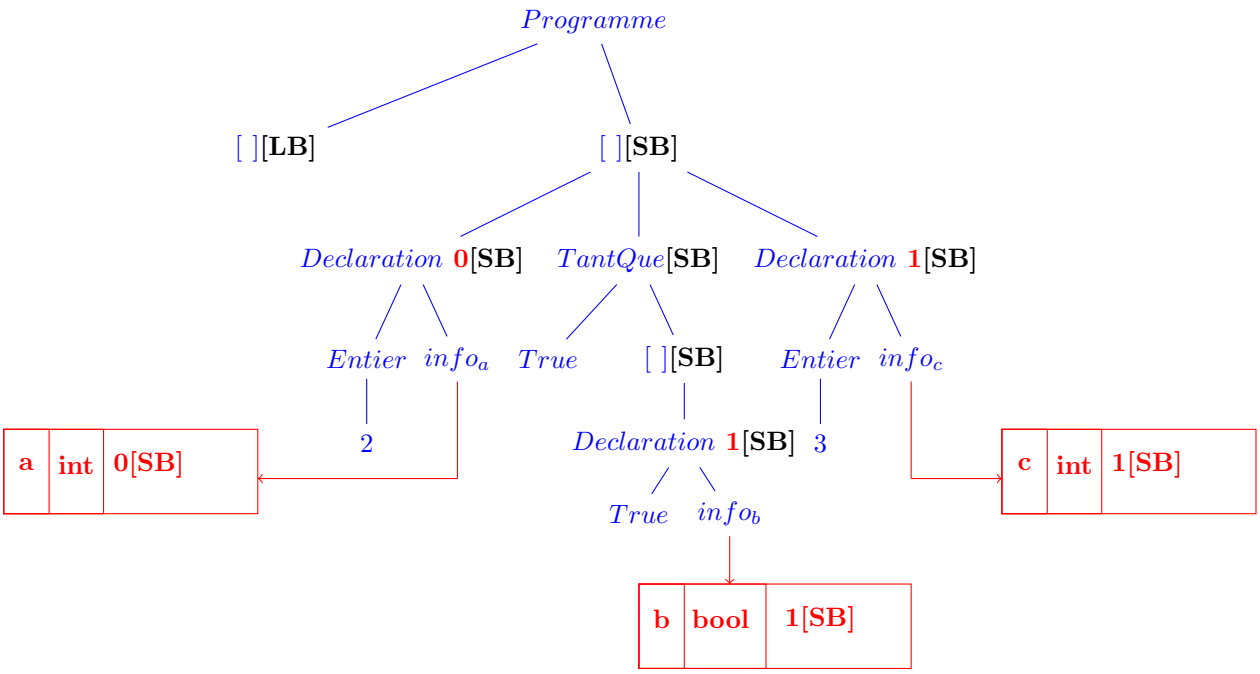
L'AstType (après la phase de typage) est le suivant :



Ce qu'on veut obtenir :

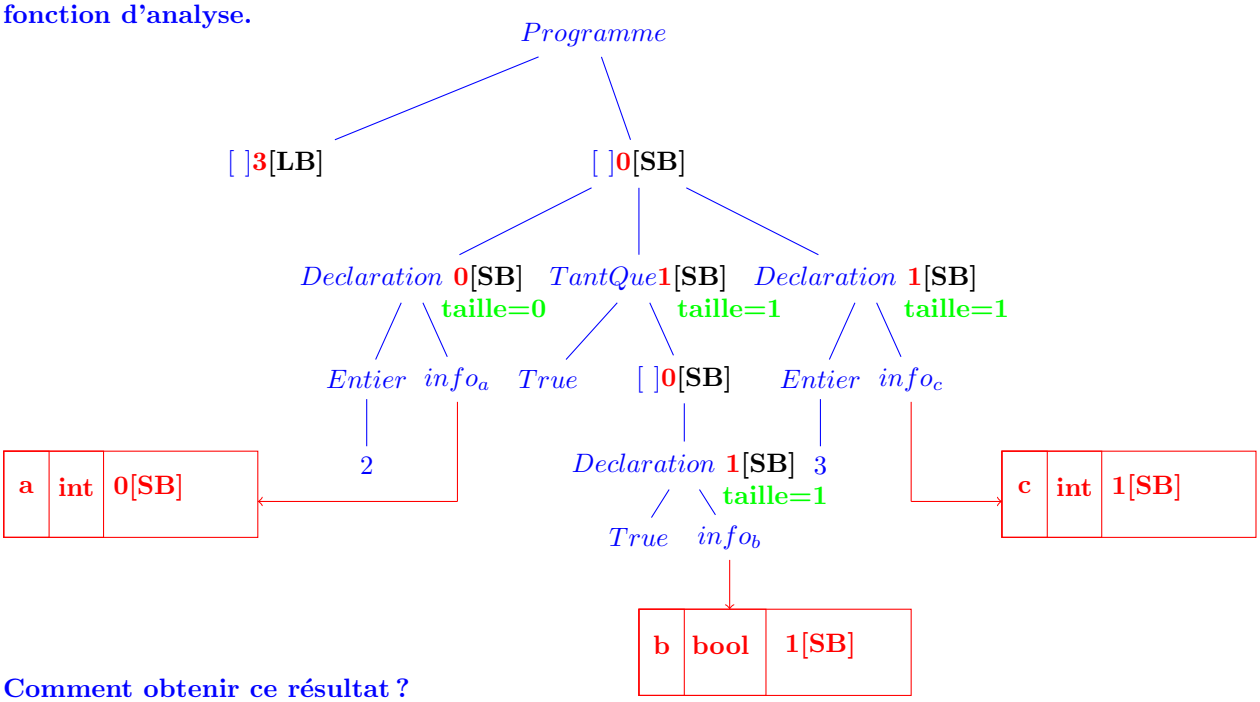


Les variables locales du programme principal et des fonctions ne sont pas placées en référence au même registre → il faut descendre le registre dans l'arbre → paramètre de la fonction d'analyse.



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)
```

Traduction des langages

Génération de code

Objectif :

— Définir les actions à réaliser par la passe de génération de code

1 De RAT à TAM

- **Cours transparents 135 à 145 pour pouvoir faire notre compilateur, transparents 146-166 en complément.**
- **Le point important à ne pas oublier est que l'on réalise un *compilateur*, un traducteur du langage RAT au langage TAM. On ne cherche pas à évaluer le code source, on cherche à générer un code TAM qui, si on l'exécute, a le même effet que celui défini par le code source.**
- **La machine TAM est une machine à pile (cf résumé de la machine TAM sur moodle) : toutes les instructions prennent des arguments en sommet de pile, les consomment, et déposent leur résultat. par Exemple pour exécuter $4+6$, il faut : empiler 4 ; empiler 6 ; appeler l'opération prédéfinie IAdd. L'opération IAdd dépile ses deux arguments et empile le résultat.**
- **La solution (détaillée ensuite) va consister à traduire individuellement chaque instruction/construction et à concaténer les traductions élémentaires. On ne cherche pas à faire du code intelligent ou optimisé, ça serait une étape ultérieure (que nous ne ferons pas faute de temps).**

▷ **Exercice 1** *Donner le code TAM correspondant au code RAT suivant :*

```
prog {
  const a = 8;
  rat x = [6/a];
  int y = (a+1);
  x = (x + [3/2]);
  while (y < 12) {
    rat z = (x * [5/y]);
    print z;
    y = ( y + 1);
  }
}
```

```
prog {
  const a = 8;
  rat x = [6/a];
  int y = (a+1);
  x = (x + [3/2]);
  while (y < 12) {
    rat z = (x * [5/y]);
    print z;
    y = ( y + 1);
```

```
  }
}
```

Le code a générer est :

```
JUMP main          on commence par exécuter le main
pgcd               pgcd (a,b) (a en -2[LB], b en -1[LB])
...                entête du fichier

main
PUSH 2             la place pour un rationnel
LOADL 6            on charge la valeur du numérateur
LOADL 8            on récupère la valeur de la constante a
CALL (SB) norm     on normalise le rationel
STORE (2) 0[SB]    on range la valeur de x ( 2 : taille / 0 : déplacement)
PUSH 1             la place pour un entier
LOADL 8            on récupère a
LOADL 1            on charge 1
SUBR IAdd          on réalise l'addition
STORE (1) 2[SB]    on range la valeur de y ( 1 : taille / 2 : déplacement)
LOAD (2) 0[SB]     on récupère x (2 : taille / 0 : déplacement)
LOADL 3            on charge la numérateur
LOADL 2            on charge le dénumérateur
CALL (SB) RAdd     addition de rationnel ([n1/d1]+[n2/d2] = [ (n1*d2) + (n2 * d1) / (d1 * d2) ])
STORE (2) 0[SB]    on range la valeur de x ( 2 : taille / 0 : déplacement)
etiq0              debut while
LOAD (1) 2[SB]     on charge la valeur de y ( 1 : taille / 2 : déplacement)
LOADL 12           on charge la constante 12
SUBR ILss          on compare les deux éléments de tête de pile
JUMPIF (0) etiq1   si la condition n'est pas validée on à fini la while
PUSH 2             la place pour un rationnel
LOAD (2) 0[SB]     on charge la valeur de x ( 2 : taille / 0 : déplacement)
LOADL 5            on charge la constante 5
LOAD (1) 2[SB]     on charge la valeur de y ( 1 : taille / 2 : déplacement)
CALL (SB) RMul     on réalise la multiplication
STORE (2) 3[SB]    on range la valeur de z (2 : taille / 3 :déplacement)
LOAD (2) 3[SB]     on charge la valeur de z (2 : taille / 3 :déplacement
CALL (SB) ROut     on affiche la valeur de z
LOAD (1) 2[SB]     on charge la valeur de y ( 1 : taille / 2 : déplacement)
LOADL 1            on charge la constante 1
SUBR IAdd          on réalise l'addition
STORE (1) 2[SB]    on range la valeur de y ( 1 : taille / 2 : déplacement)
POP (0) 2          on libère la place des variables
JUMP etiq0         on reboucle
etiq1              fin while
POP (0) 3          on libère la place des variables
HALT               fin
```

2 Passe de génération de code

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 dernière passe est une passe de génération de code. Il n'y a donc plus d'AST à générer.

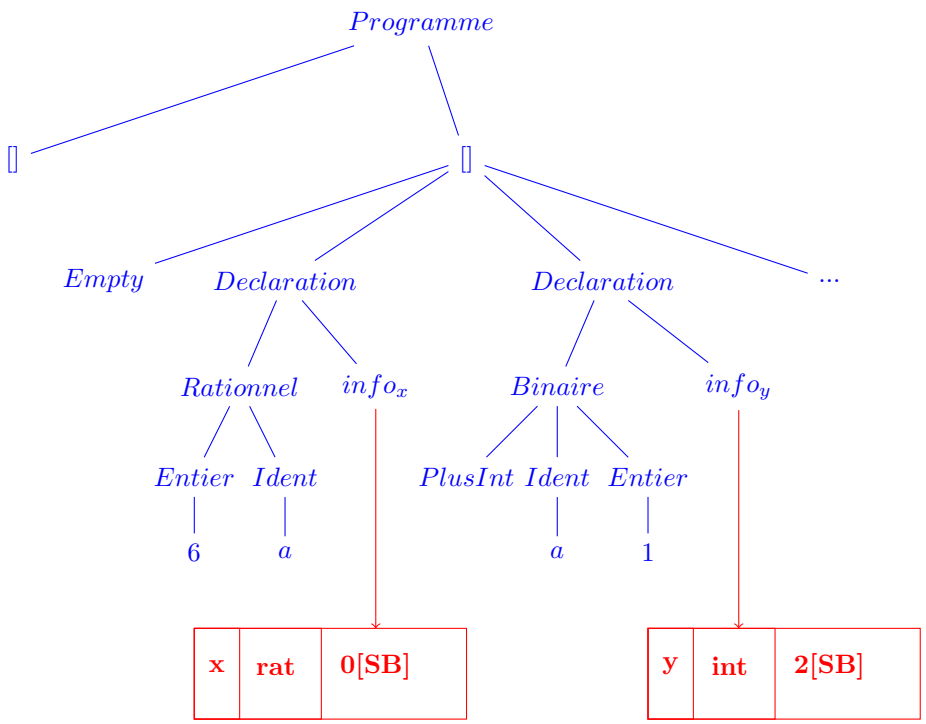
▷ **Exercice 2** *Définir les actions à réaliser lors de la passe de génération de code.*

On prend un extrait de l'exemple précédent :

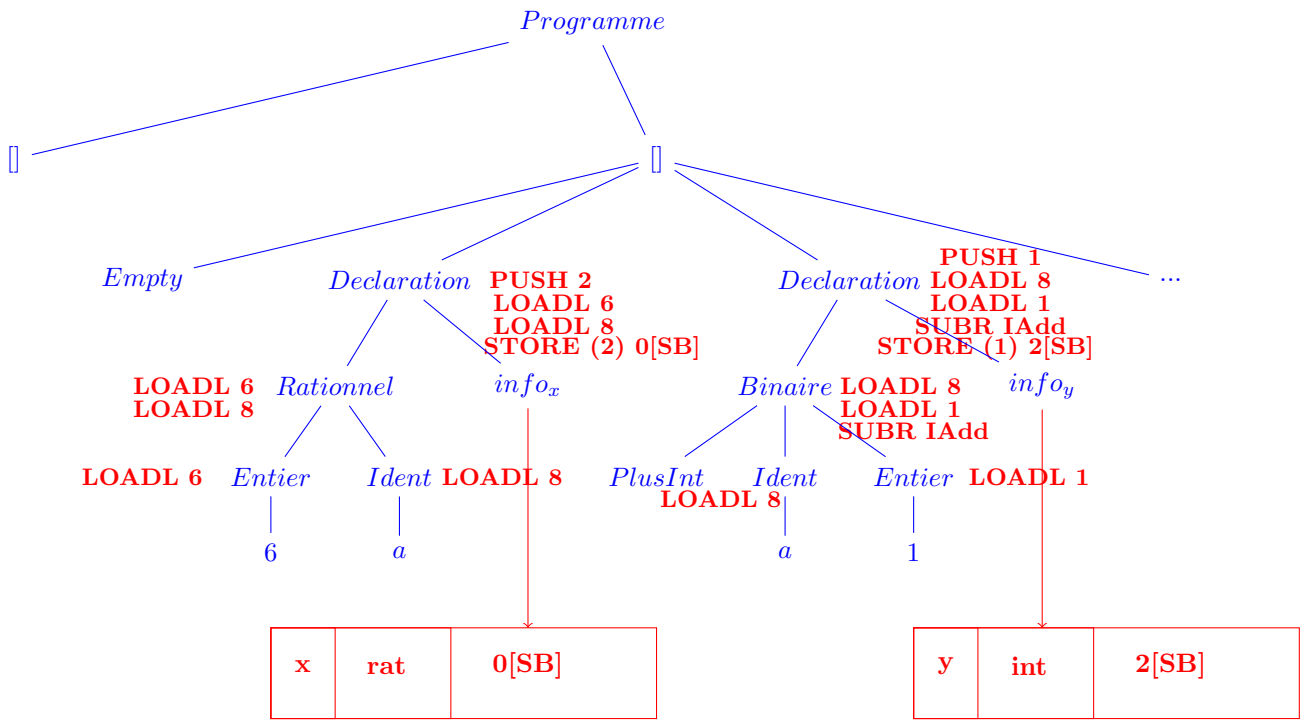
```
prog {
  const a = 8;
  rat x = [6/a];
  int y = (a+1);
 ...
}
```

On a : @x=0[SB], @y=2[SB], ...

L'AstDep (après la phase de déplacement) est le suivant :



On annote l'arbre pour montrer ce qu'on veut :



On note que le code remonte dans l'arbre → paramètre de retour de la fonction d'analyse.

Comment obtenir ce résultat ? Le code remonte de la fonction d'analyse. C'est la dernière passe, nous ne construisons plus d'arbre.

Commençons par le bas : analyse des expressions, puis analyse des instructions, puis analyse des fonctions et du programme principal.

```
(* expression -> string *)
(* Produit le code correspondant à l'instruction. L'exécution de ce code laissera
 * en sommet de pile le résultat de l'évaluation de cette expression. *)
let rec analyse_code_expression e =
  match e with
  | AppelFonction (info,le) ->
    - concaténer le code correspondant à chacune des expressions arguments
      [ chaque code laisse en sommet de pile la valeur correspondant.
        En concaténant, on obtient dans la pile la liste des arguments ]
    - générer l'appel à la fonction : CALL (ST) n
  | Rationnel (e1,e2) ->
    - concaténer le code pour e1 et pour e2
  | Numerateur e1 ->
    - code pour e1, suivi de l'élimination du dénominateur :
      (analyse_code_expression e1) ^ "POP␣(0)␣1\n"
  | Denominateur e1 ->
    - code pour e1, suivi de l'élimination du numérateur :
```

```
    (analyse_code_expression e1) ^ "POP (1) 1\n"
  | Ident info ->
    - ( info_ast_to_info info) doit être un InfoVar(t,dep,reg) ou un InfoConst(v)
      [ erreur interne sinon ]
    - variable : LOAD (taille du type) dép[reg]
    - constante : LOADL v
  | True -> "LOADL 1\n"
  | False -> "LOADL 0\n"
  | Entier i -> "LOADL " ^ (string_of_int i) ^ "\n"
  | Binaire (b,e1,e2) ->
    - générer le code pour e1
    - générer le code pour e2
    - les concaténer, suivi de l'appel à la bonne opération (SUBR IAdd, SUBR IMul,
      SUBR ILss, SUBR IEq, SUBR BEq, CALL (ST) RAdd, CALL (ST) RMUL)
      [ l'addition et la multiplication de rationnels ne sont pas des opérations TAM,
        on appelle des fonctions auxiliaires )

(* instruction -> -> string *)
let rec analyse_code_instruction i =
  match i with
  | Declaration (e, info) ->
    - ( info_ast_to_info info) doit être un InfoVar (typ, dép, reg).
    - concaténer le code pour réserver la place (PUSH taille du type),
      le code correspondant à l'expression e,
      le rangement du résultat dans la variable (STORE (taille du type) dép[reg])
  | Affectation (e, info) ->
    - ( info_ast_to_info info) doit être un InfoVar (typ, dép, reg).
    - concaténer le code correspondant à l'expression e,
      le rangement du résultat dans la variable (STORE (taille du type) dép[reg])
  | AffichageInt e ->
      (analyse_code_expression e) ^ "SUBR IOut\n"
  | AffichageRat e ->
      (analyse_code_expression e ^ "CALL (ST) ROut\n"
  | AffichageBool e ->
      (analyse_code_expression e) ^ "SUBR BOut\n"
  | Conditionnelle (cond, then , else) ->
      - code à produire : code de c, saut conditionnel vers label else (JUMPIF),
        code du then + saut àla fin , label du else, code du else, label de fin
      - utiliser getEtiquette() pour obtenir une nouvelle étiquette , distincte à chaque appel.
      - code_de_cond
        ^ "JUMPIF (0) " ^ lelse ^ "\n"
        ^ code_de_then
        ^ "JUMP " ^ lfinelse ^ "\n"
        ^ lelse ^ "\n"
        ^ code_de_else
        ^ lfinelse ^ "\n"
  | TantQue (c,b) ->
      - code à produire : label de début, code de c, saut conditionnel à la fin ,
        code de b, saut au début
  | Empty -> ""


and analyse_code_bloc li =
```

```
(* - déterminer la taille occupée par les variables locales de ce bloc
     (il peut être utile d'introduire une fonction auxiliaire qui donne
     la taille occupée par une instruction : Decl => taille du type / autre => 0)
   - générer le code pour la liste d'instructions
     suivi de la libération des variables locales (POP (0) taillevarloc) *)
  let taille = List.fold_right (fun i ti -> (taille_variables_declarees i) + ti) li 0 in
  let popfinal = "POP (0) " ^ (string_of_int taille) ^ "\n" in
  (analyse_code_li li) ^ popfinal

(* une liste d'instruction est un bloc dont on ignore la taille des variables locales *)
and analyse_code_li li =
  String.concat "" (List.map analyse_code_instruction li)

(* AstPlacement.fonction -> string *)
let analyse_code_fonction (Fonction(info, _, li, e)) =
  - Attention : li n'est pas un bloc (on peut utiliser les var locales dans e)
  - info doit être InfoFun(nom, typeRet, typeParams)
  - déterminer la taille des variables locales (comme ci-dessus)
  - déterminer la taille occupée par les paramètres (somme des tailles de typeParams)
  - générer le code avec :
    le nom (qui servira de label)
    + le code des instructions
    + le code de e
    + la libération des var locales : POP (taille typeret) taillevarloc
    + RETURN (taille typeret) tailleparam
  - explication des libérations (transparent 128) :
    - e laisse en sommet de pile le résultat de la fonction, et en dessous se trouvent
      les variables locales que l'on élimine via le POP
    - on a maintenant (de haut en bas) : la valeur de retour, l'enregistrement d'activation,
      les paramètres. RETURN va restaurer le compteur ordinal et libérer
      l'enregistrement et les paramètres.

Le code d'un Programme(fonctions,prog) est la concaténation du code de la
bibliothèque (getEntete) + code des fonctions + label "main" + code du bloc
prog + HALT.
```

Traduction des langages

Typage

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

1 Rappel : le typage du langage Rat

1.1 Grammaire du langage RAT

1. $PROG' \rightarrow PROG\$$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ \{IS\ return\ E\ ;\}$
4. $PROG \rightarrow\ id\ \ BLOC$
5. $BLOC \rightarrow\ \{\ IS\ \}$
6. $IS \rightarrow\ I\ \ IS$
7. $IS \rightarrow \Lambda$
8. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow\ id\ =\ E\ ;$
10. $I \rightarrow\ const\ \ id\ =\ entier\ ;$
11. $I \rightarrow\ print\ \ E;$
12. $I \rightarrow\ if\ \ E\ \ BLOC\ \ else\ \ BLOC$
13. $I \rightarrow\ while\ \ E\ \ BLOC$
14. $DP \rightarrow \Lambda$
15. $DP \rightarrow TYPE\ id\ DP$
16. $TYPE \rightarrow\ bool$
17. $TYPE \rightarrow\ int$
18. $TYPE \rightarrow\ rat$
19. $E \rightarrow call\ id\ (\ CP\ )$
20. $CP \rightarrow \Lambda$
21. $CP \rightarrow E\ CP$
22. $E \rightarrow\ [\ E\ /\ E\ ]$
23. $E \rightarrow\ num\ \ E$
24. $E \rightarrow\ denom\ \ E$
25. $E \rightarrow\ id$
26. $E \rightarrow\ true$
27. $E \rightarrow\ false$
28. $E \rightarrow\ entier$
29. $E \rightarrow\ (\ E\ +\ \ E\ )$
30. $E \rightarrow\ (\ E\ *\ \ E\ )$
31. $E \rightarrow\ (\ E\ =\ \ E\ )$
32. $E \rightarrow\ (\ E\ <\ \ E\ )$
33. $E \rightarrow\ (\ E\ )$

1.2 Jugements de typage du langage RAT

Axiomes

- $(x, \tau) :: \sigma \vdash x : \tau$
- $\dfrac{\sigma \vdash x : \tau}{(y, \tau') :: \sigma \vdash x : \tau}$
- $\sigma \vdash true : bool$
- $\sigma \vdash false : bool$
- $\sigma \vdash entier : int$

Expression

— $\dfrac{\sigma \vdash E_1 : int \qquad \sigma \vdash E_2 : int}{\sigma \vdash [\ E_1\ /\ E_2\ ] : rat}$

— $\dfrac{\sigma \vdash E : rat}{\sigma \vdash num\ E : int}$

— $\dfrac{\sigma \vdash E : rat}{\sigma \vdash denom\ E : int}$

— $\dfrac{\sigma \vdash E : \tau}{\sigma \vdash (E) : \tau}$

— $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 + E_2) : int}$

— $\dfrac{\sigma \vdash E_1 : rat \quad \sigma \vdash E_2 : rat}{\sigma \vdash (E_1 + E_2) : rat}$

— $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 * E_2) : int}$

— $\dfrac{\sigma \vdash E_1 : rat \quad \sigma \vdash E_2 : rat}{\sigma \vdash (E_1 * E_2) : rat}$

— $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 = E_2) : bool}$

— $\dfrac{\sigma \vdash E_1 : bool \quad \sigma \vdash E_2 : bool}{\sigma \vdash (E_1 = E_2) : bool}$

— $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 < E_2) : bool}$

— On se limitera à ces signatures.

Structures de contrôle

— $\dfrac{\sigma \vdash E : bool \qquad \sigma \vdash BLOC_1 : void \qquad \sigma \vdash BLOC_2 : void}{\sigma \vdash if\ E\ BLOC_1\ else\ BLOC_2 : void, []}$

— $\dfrac{\sigma \vdash E : bool \qquad \sigma \vdash BLOC : void}{\sigma \vdash while\ E\ BLOC : void, []}$

Déclaration / affectation

— $\dfrac{\sigma \vdash TYPE : \tau \qquad \sigma \vdash E : \tau}{\sigma \vdash TYPE\ id = E : void, [id, \tau]}$

— $\dfrac{\sigma \vdash id : \tau \qquad \sigma \vdash E : \tau}{\sigma \vdash id = E : void, []}$

Autres instructions

— $\dfrac{}{\sigma \vdash const\ \ id\ =\ entier : void, [id, int]}$

— $\dfrac{\sigma \vdash E : \tau}{\sigma \vdash print\ \ E : void, []}$

Déclaration de fonction

— $\dfrac{A \quad B \quad C \quad D}{\sigma \vdash TYPE\ id\ (DP)\ \{IS\ return\ E;\} : void, [id, \tau_2 \to \tau_1]}$

— A : $\sigma \vdash TYPE : \tau_1$
— B : $\sigma \vdash DP : \tau_2, \sigma_p \qquad (DP = TYPE_1\ id_1 \ldots TYPE_n\ id_n)$
— C : $(id, \tau_2 \to \tau_1)@\sigma_p@\sigma \vdash IS : void, \sigma_l$
— D : $(id, \tau_2 \to \tau_1)@\sigma_l@\sigma_p@\sigma \vdash E : \tau_1$

— $\dfrac{\sigma \vdash TYPE_1 : \tau_1 \quad \ldots \quad TYPE_n : \tau_n}{\sigma \vdash TYPE_1\ id_1\ \ldots\ TYPE_n\ id_n : \tau_1 \times \cdots \times \tau_n, [(id_1, \tau_1); \ldots; (id_n, \tau_n)]}$

Appel de fonction

— $\dfrac{\sigma \vdash id : \tau_1 \to \tau_2 \qquad E_1 \ \ldots \ E_n : \tau_1}{\sigma \vdash call\ id\ (E_1 \ \ldots \ E_n) : \tau_2}$

— $\dfrac{\sigma \vdash E_1 : \tau_1 \quad \ldots \quad E_n : \tau_n}{\sigma \vdash E_1 \ \ldots \ E_n : \tau_1 \times \cdots \times \tau_n}$

Suite d'instructions

— $\dfrac{\sigma \vdash IS : void, \sigma'}{\sigma \vdash \{IS\} : void}$

— $\dfrac{\sigma \vdash I : void, \sigma' \qquad \sigma' @ \sigma \vdash IS : void, \sigma''}{\sigma \vdash I\ IS : void, \sigma'' @ \sigma'}$

— $\sigma \vdash \quad : void$

Le programme

— $\dfrac{\sigma \vdash FUN : void, \sigma' \qquad \sigma' @ \sigma \vdash PROG : void, \sigma''}{\sigma \vdash FUN\ PROG : void, \sigma'' @ \sigma'}$

— $\dfrac{\sigma \vdash BLOC : void}{\sigma \vdash id\ BLOC : void}$

2 Passe de typage

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 seconde passe est une passe de typage. C'est elle qui vérifie la conformité des types déclarés et associe aux identifiants leurs informations de type.

2.1 Structure de l'AST après la passe de typage

La passe de typage réalise des vérifications de type qui nécessitent une mise à jour des informations de type de identificateurs. Elle prépare également les passes suivantes, par exemple en choisissant la "version" de l'opérateur à utiliser en cas de surcharge des opérateurs.

▷ **Exercice 1** *Définir la structure de l'AST après la passe de typage.*

- **Les informations des identifiants seront mises à jour avec les informations de type.**
- **Les informations de types n'ont plus besoin d'être conservées dans les nœuds de l'arbre.**
- **Les opérateurs surchargés sont "résolus" et remplacés par des opérateurs non surchargés.**

Voici un type possible pour l'arbre après la passe de typage :

```
(* Opérateurs binaires existants dans Rat - résolution de la surcharge *)
type binaire = PlusInt | PlusRat | MultInt | MultRat | EquInt | EquBool | Inf
```

```
(* Expressions existantes dans Rat *)
(* = expression de AstTds *)
type expression =
  | AppelFonction of Tds.info_ast * expression list
  | Rationnel of expression * expression
  | Numerateur of expression
  | Denominateur of expression
  | Ident of Tds.info_ast
  | True
  | False
  | Entier of int
  | Binaire of binaire * expression * expression

(* instructions existantes Rat *)
(* = instruction de AstTds + informations associées aux identificateurs, mises à jour *)
(* + résolution de la surcharge de l'affichage *)
type bloc = instruction list
 and instruction =
  | Declaration of expression * Tds.info_ast
  | Affectation of expression * Tds.info_ast
  | AffichageInt of expression
  | AffichageRat of expression
  | AffichageBool of expression
  | Conditionnelle of expression * bloc * bloc
  | TantQue of expression * bloc
  | Empty (* les nœuds ayant disparus: Const *)

(* informations associées à l'identificateur (dont son nom), liste des paramètres, corps, expression de retour *)
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 typage

La passe de typage est donc une transformation d'un arbre vers un autre.

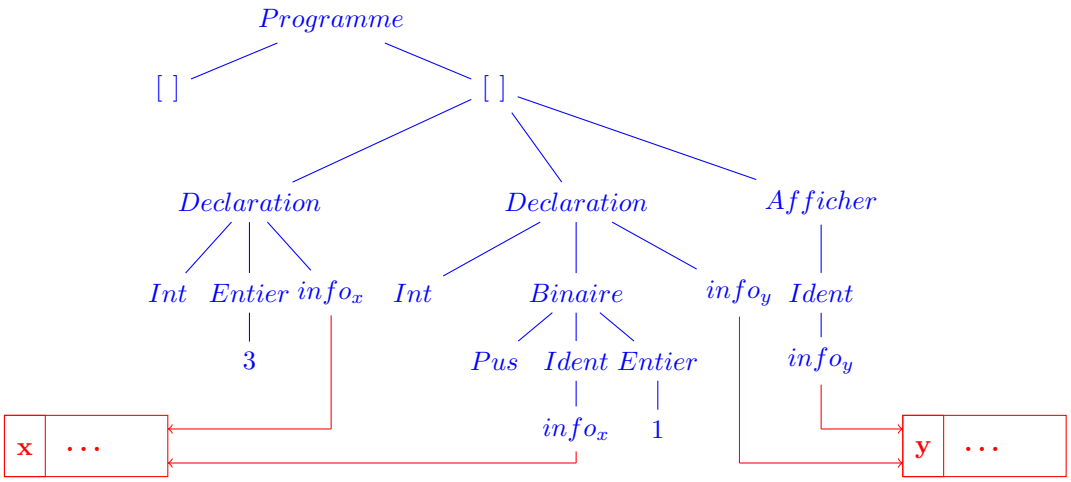
▷ **Exercice 2** *Définir les actions à réaliser lors de la passe de typage.*

On reprend l'exemple :

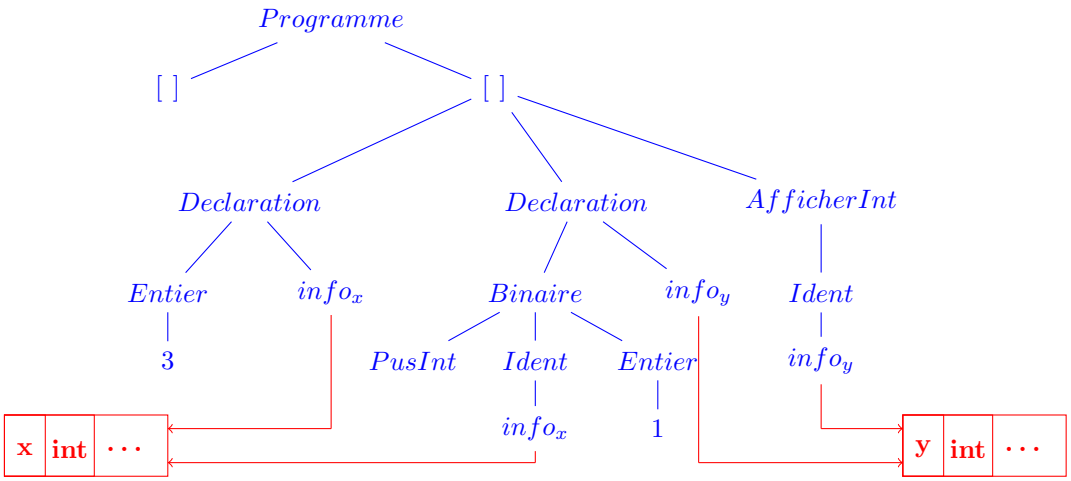
1. **Code :**

```
main{
 int x = 3;
 int y = x+1;
 print y;
}
```

2. **AstTds**



3. **Actions à réaliser :**
   (a) **Mettre à jour $info_x$ avec le type de x**
   (b) **Vérifier que + est utilisé avec une bonne signature**
   (c) **Identifier la version de + à utiliser (surcharge)**
   (d) **Vérifier que le type de retour de + est compatible avec le type déclaré de y**
   (e) **Mettre à jour $info_y$ avec le type de y**
   (f) **Identifier la version d'affichage à utiliser (surcharge)**
4. **AstType**



Comment obtenir ce résultat ? Les informations de type sont connues au niveau des feuilles, il faut donc que les informations de types remontent dans l'arbre ⇒ valeur de retour des fonctions d'analyse. Nous n'en avons besoin que sur les expressions, car les instructions sont typées en unit / void.

Commençons par l'analyse des instructions puis des expressions. L'analyse du programme principal et l'analyse des fonctions ne sont pas détaillées et sont à définir.

```
let rec analyse_type_instruction i =
  match i with
  | AstTds.Declaration (t, e, info) ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - vérifier que te est compatible avec t [ sinon TypeInattendu ]
        - mettre à jour le type dans info (modifier_type_info t info);
        - retourner Declaration (ne, info)
  | AstTds.Affectation (e, info) ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - vérifier que te est compatible avec le type dans l'info
        - retourner Affectation (ne, info)
    (* Affichage *)
    | AstTds.Affichage e ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - selon te, construire le nouvel arbre en résolvant la surcharge
    (*Conditionnelle *)
    | AstTds.Conditionnelle (c,t,e) ->
        - analyser c pour obtenir son type (tc) et le nouvel arbre (nc)
        - vérifier que tc est Bool [ sinon TypeInattendu ]
        - analyser les blocs t et e
        - retourner Conditionnelle(nc,nt,ne)
    (* Tant Que *)
    | AstTds.TantQue (c,b) ->
      - analyser c pour obtenir son type (tc) et le nouvel arbre (nc)
      - vérifier que tc est Bool [ sinon TypeInattendu ]
      - analyser le bloc b
      - retourner TantQue(nc,nb)
    | AstTds.Empty -> Empty

and analyse_type_bloc b = map analyse_type_instruction b
```

Puis on passe aux expressions :

```
(* Asttds.ast -> (ast,typ) *)
let rec analyse_type_expression e =
  match e with
  | AstTds.AppelFonction (info,le) ->                    (* Appel de fonction *)
    - ( info_ast_to_info info) doit être un InfoFun(nom,typeRet,typeParams)
      [ sinon erreur interne, ça doit être garanti par la passe précédente ]
    - analyser chacune des expressions dans le, pour obtenir nle (nouvelle
      liste des arguments) et ltype ( liste des types de ces expressions)
    - vérifier que ltype est compatible avec typeParams
     [ sinon exception TypesParametresInattendus ]
    - retourner le couple (AppelFonction (info,nle), typeRet)
  | AstTds.Rationnel (e1,e2) ->                          (* Définition d'un rationnel *)
    - analyser e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
    - analyser e2 pour obtenir ne2 (nouvel arbre) et t2 (type calculé)
    - vérifier que t1 et t2 sont Int (sinon TypeInattendu)
    - retourner (Rationnel(ne1,ne2), Rat)
  | AstTds.Numerateur e1 ->                              (* Accès au numérateur *)
    - analyser l'expression e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
    - vérifier que t1 est Rat (sinon TypeInattendu)
```

```
  - retourner (Numerateur ne1, Int)
| AstTds.Denominateur e1 ->                          (* Accès au dénominateur *)
  - analyser l'expression e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
  - vérifier que t1 est Rat (sinon TypeInattendu)
  - retourner (Denominateur ne1, Int)
 | AstTds.Ident info ->                              (* Accès à un identifiant *)
  begin
    match info_ast_to_info info  with
    | InfoVar (_,t,_,_) -> (Ident info, t)
    | InfoConst (_,_) -> (Ident info, Int)
    | _ -> failwith ("Internal_error:_symbol_not_found")
  end
| AstTds.Binaire (b,e1,e2) ->                        (* Opération binaire *)
  - analyser e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
  - analyser e2 pour obtenir ne2 (nouvel arbre) et t2 (type calculé)
  - identifier les cas corrects et retourner le couple (arbre,type) correspondant:
    - Int, Plus, Int -> (Binaire (PlusInt,ne1,ne2), Int)
    - Rat, Plus, Rat -> (Binaire (PlusRat,ne1,ne2), Rat)
    - Int, Equ, Int  -> (Binaire (EquInt,ne1,ne2), Bool)
    - Bool, Equ, Bool -> (Binaire (EquBool,ne1,ne2), Bool)
    - etc
    - sinon TypeBinaireInattendu
| AstTds.True -> (True, Bool)
| AstTds.False -> (False, Bool)
| AstTds.Entier i -> (Entier i, Int)
```

Il reste à définir l'analyse du programme principal et l'analyse des fonctions.

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.

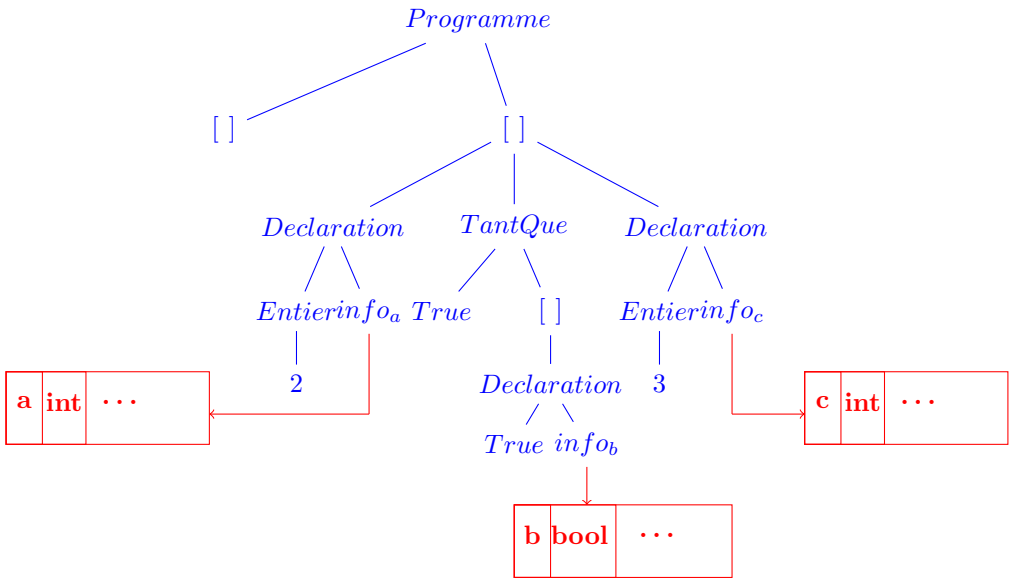
▷ **Exercice 3** *Définir les actions à réaliser lors de la passe de placement mémoire.*

On prend un exemple simple :

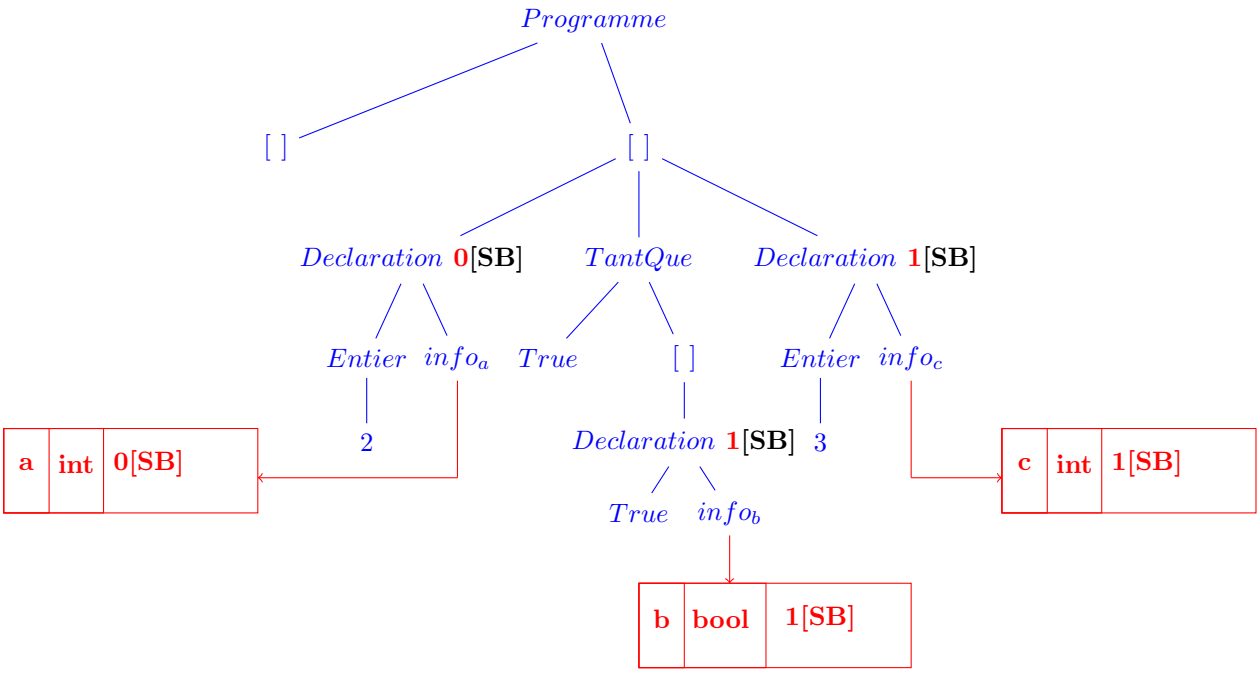
```
dep {
  int a = 2;
  while(true){
    bool b = true;
    }
  int c = 3;
}
```

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

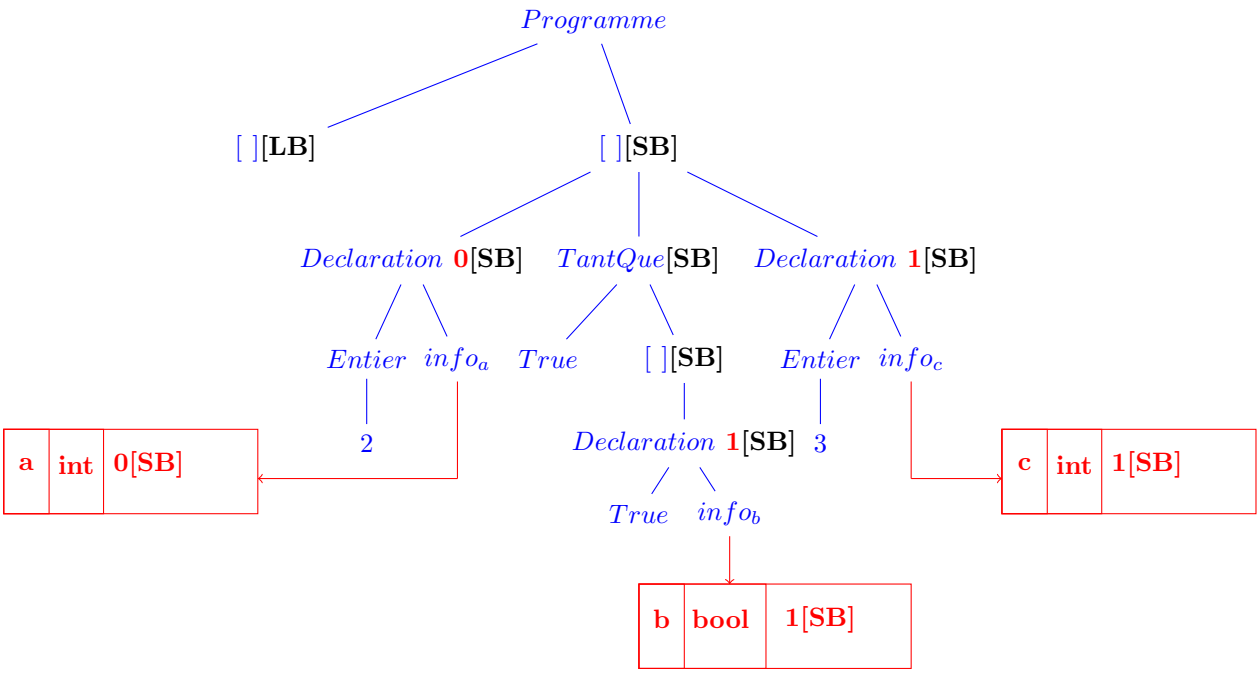
L'AstType (après la phase de typage) est le suivant :



Ce qu'on veut obtenir :

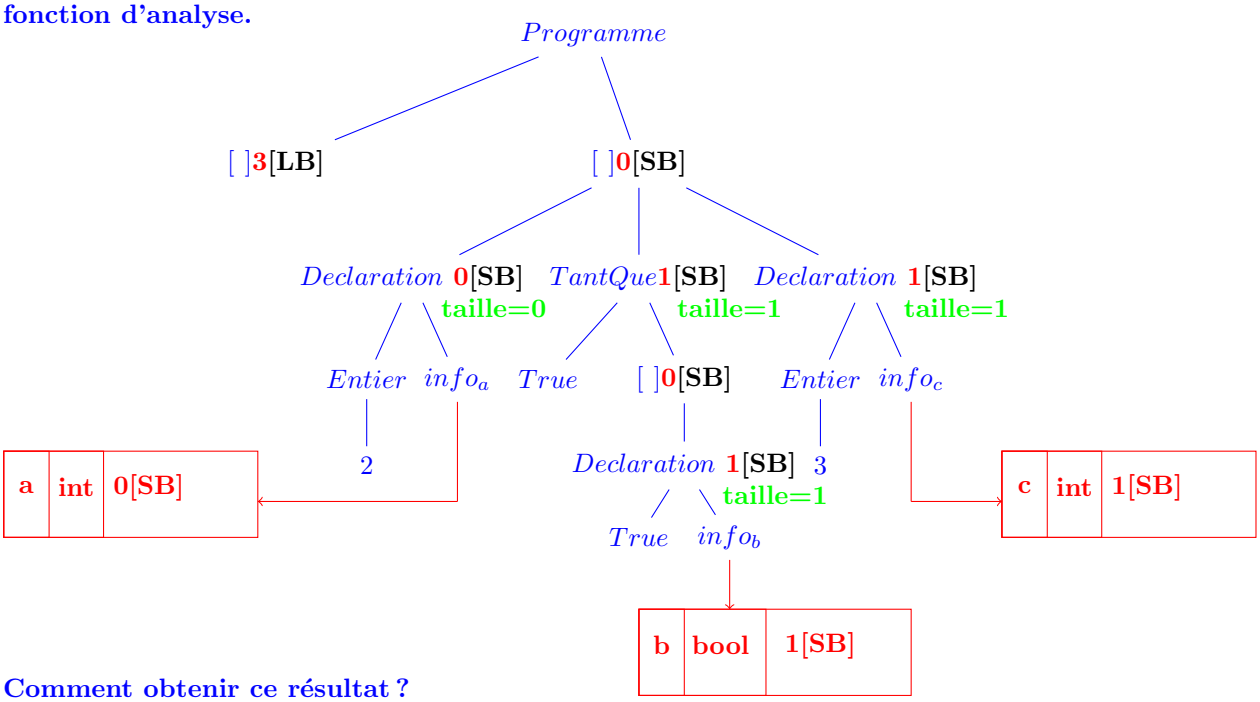


Les variables locales du programme principal et des fonctions ne sont pas placées en référence au même registre → il faut descendre le registre dans l'arbre → paramètre de la fonction d'analyse.



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)
```

Traduction des langages

Génération de code

Objectif :

— Définir les actions à réaliser par la passe de génération de code

1 De RAT à TAM

- **Cours transparents 135 à 145 pour pouvoir faire notre compilateur, transparents 146-166 en complément.**
- **Le point important à ne pas oublier est que l'on réalise un *compilateur*, un traducteur du langage RAT au langage TAM. On ne cherche pas à évaluer le code source, on cherche à générer un code TAM qui, si on l'exécute, a le même effet que celui défini par le code source.**
- **La machine TAM est une machine à pile (cf résumé de la machine TAM sur moodle) : toutes les instructions prennent des arguments en sommet de pile, les consomment, et déposent leur résultat. par Exemple pour exécuter $4+6$, il faut : empiler 4 ; empiler 6 ; appeler l'opération prédéfinie IAdd. L'opération IAdd dépile ses deux arguments et empile le résultat.**
- **La solution (détaillée ensuite) va consister à traduire individuellement chaque instruction/construction et à concaténer les traductions élémentaires. On ne cherche pas à faire du code intelligent ou optimisé, ça serait une étape ultérieure (que nous ne ferons pas faute de temps).**

▷ **Exercice 1** *Donner le code TAM correspondant au code RAT suivant :*

```
prog {
  const a = 8;
  rat x = [6/a];
  int y = (a+1);
  x = (x + [3/2]);
  while (y < 12) {
    rat z = (x * [5/y]);
    print z;
    y = ( y + 1);
  }
}
```

```
prog {
  const a = 8;
  rat x = [6/a];
  int y = (a+1);
  x = (x + [3/2]);
  while (y < 12) {
    rat z = (x * [5/y]);
    print z;
    y = ( y + 1);
```

```
  }
}
```

Le code a générer est :

```
JUMP main          on commence par exécuter le main
pgcd               pgcd (a,b) (a en -2[LB], b en -1[LB])
...                entête du fichier

main
PUSH 2             la place pour un rationnel
LOADL 6            on charge la valeur du numérateur
LOADL 8            on récupère la valeur de la constante a
CALL (SB) norm     on normalise le rationel
STORE (2) 0[SB]    on range la valeur de x ( 2 : taille / 0 : déplacement)
PUSH 1             la place pour un entier
LOADL 8            on récupère a
LOADL 1            on charge 1
SUBR IAdd          on réalise l'addition
STORE (1) 2[SB]    on range la valeur de y ( 1 : taille / 2 : déplacement)
LOAD (2) 0[SB]     on récupère x (2 : taille / 0 : déplacement)
LOADL 3            on charge la numérateur
LOADL 2            on charge le dénumérateur
CALL (SB) RAdd     addition de rationnel ([n1/d1]+[n2/d2] = [ (n1*d2) + (n2 * d1) / (d1 * d2) ])
STORE (2) 0[SB]    on range la valeur de x ( 2 : taille / 0 : déplacement)
etiq0              debut while
LOAD (1) 2[SB]     on charge la valeur de y ( 1 : taille / 2 : déplacement)
LOADL 12           on charge la constante 12
SUBR ILss          on compare les deux éléments de tête de pile
JUMPIF (0) etiq1   si la condition n'est pas validée on à fini la while
PUSH 2             la place pour un rationnel
LOAD (2) 0[SB]     on charge la valeur de x ( 2 : taille / 0 : déplacement)
LOADL 5            on charge la constante 5
LOAD (1) 2[SB]     on charge la valeur de y ( 1 : taille / 2 : déplacement)
CALL (SB) RMul     on réalise la multiplication
STORE (2) 3[SB]    on range la valeur de z (2 : taille / 3 :déplacement)
LOAD (2) 3[SB]     on charge la valeur de z (2 : taille / 3 :déplacement
CALL (SB) ROut     on affiche la valeur de z
LOAD (1) 2[SB]     on charge la valeur de y ( 1 : taille / 2 : déplacement)
LOADL 1            on charge la constante 1
SUBR IAdd          on réalise l'addition
STORE (1) 2[SB]    on range la valeur de y ( 1 : taille / 2 : déplacement)
POP (0) 2          on libère la place des variables
JUMP etiq0         on reboucle
etiq1              fin while
POP (0) 3          on libère la place des variables
HALT               fin
```

2 Passe de génération de code

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 dernière passe est une passe de génération de code. Il n'y a donc plus d'AST à générer.

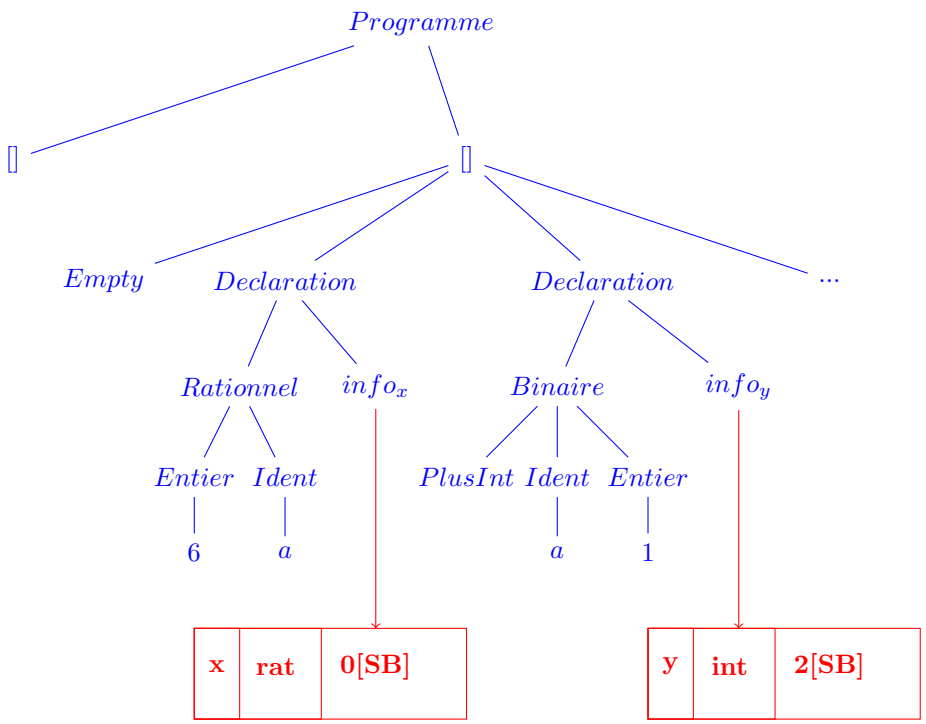
▷ **Exercice 2** *Définir les actions à réaliser lors de la passe de génération de code.*

On prend un extrait de l'exemple précédent :

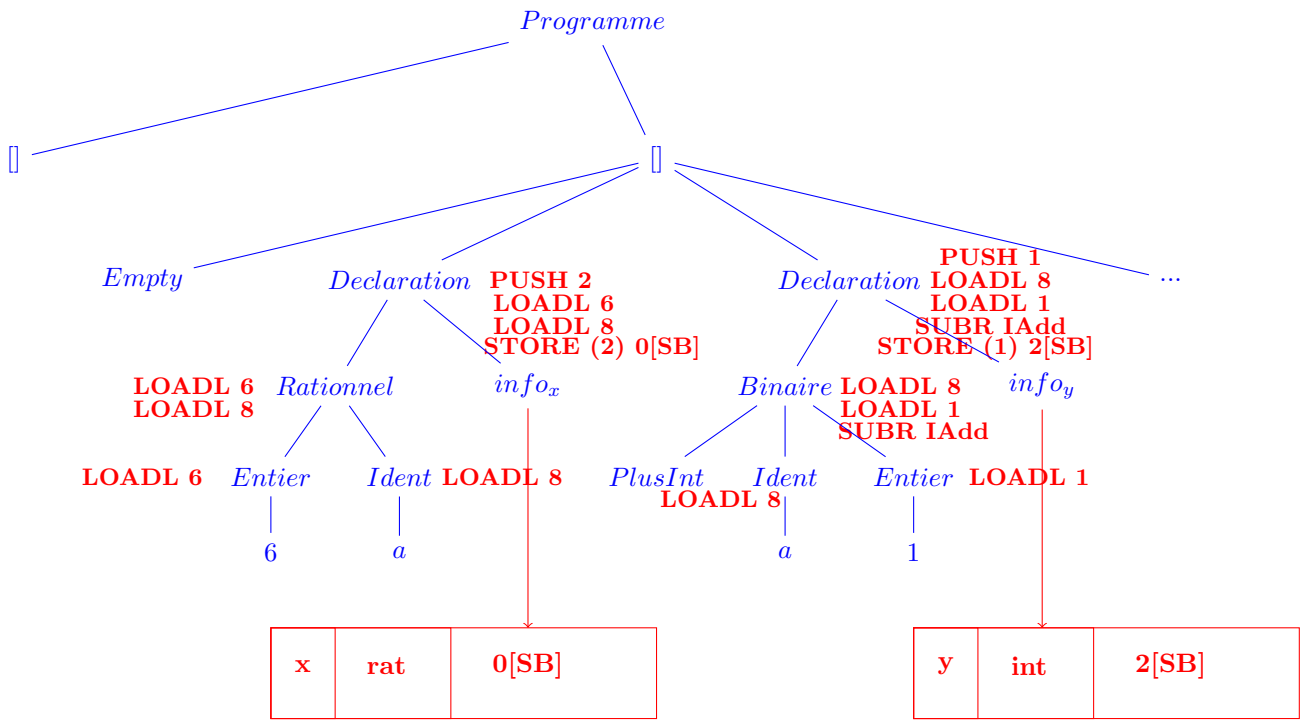
```
prog {
  const a = 8;
  rat x = [6/a];
  int y = (a+1);
 ...
}
```

On a : @x=0[SB], @y=2[SB], ...

L'AstDep (après la phase de déplacement) est le suivant :



On annote l'arbre pour montrer ce qu'on veut :



On note que le code remonte dans l'arbre → paramètre de retour de la fonction d'analyse.

Comment obtenir ce résultat ? Le code remonte de la fonction d'analyse. C'est la dernière passe, nous ne construisons plus d'arbre.

Commençons par le bas : analyse des expressions, puis analyse des instructions, puis analyse des fonctions et du programme principal.

```
(* expression -> string *)
(* Produit le code correspondant à l'instruction. L'exécution de ce code laissera
 * en sommet de pile le résultat de l'évaluation de cette expression. *)
let rec analyse_code_expression e =
  match e with
  | AppelFonction (info,le) ->
    - concaténer le code correspondant à chacune des expressions arguments
      [ chaque code laisse en sommet de pile la valeur correspondant.
        En concaténant, on obtient dans la pile la liste des arguments ]
    - générer l'appel à la fonction : CALL (ST) n
  | Rationnel (e1,e2) ->
    - concaténer le code pour e1 et pour e2
  | Numerateur e1 ->
    - code pour e1, suivi de l'élimination du dénominateur :
      (analyse_code_expression e1) ^ "POP␣(0)␣1\n"
  | Denominateur e1 ->
    - code pour e1, suivi de l'élimination du numérateur :
```

```
    (analyse_code_expression e1) ^ "POP (1) 1\n"
  | Ident info ->
    - ( info_ast_to_info info) doit être un InfoVar(t,dep,reg) ou un InfoConst(v)
      [ erreur interne sinon ]
    - variable : LOAD (taille du type) dép[reg]
    - constante : LOADL v
  | True -> "LOADL 1\n"
  | False -> "LOADL 0\n"
  | Entier i -> "LOADL " ^ (string_of_int i) ^ "\n"
  | Binaire (b,e1,e2) ->
    - générer le code pour e1
    - générer le code pour e2
    - les concaténer, suivi de l'appel à la bonne opération (SUBR IAdd, SUBR IMul,
      SUBR ILss, SUBR IEq, SUBR BEq, CALL (ST) RAdd, CALL (ST) RMUL)
      [ l'addition et la multiplication de rationnels ne sont pas des opérations TAM,
        on appelle des fonctions auxiliaires )

(* instruction -> -> string *)
let rec analyse_code_instruction i =
  match i with
  | Declaration (e, info) ->
    - ( info_ast_to_info info) doit être un InfoVar (typ, dép, reg).
    - concaténer le code pour réserver la place (PUSH taille du type),
      le code correspondant à l'expression e,
      le rangement du résultat dans la variable (STORE (taille du type) dép[reg])
  | Affectation (e, info) ->
    - ( info_ast_to_info info) doit être un InfoVar (typ, dép, reg).
    - concaténer le code correspondant à l'expression e,
      le rangement du résultat dans la variable (STORE (taille du type) dép[reg])
  | AffichageInt e ->
      (analyse_code_expression e) ^ "SUBR IOut\n"
  | AffichageRat e ->
      (analyse_code_expression e ^ "CALL (ST) ROut\n"
  | AffichageBool e ->
      (analyse_code_expression e) ^ "SUBR BOut\n"
  | Conditionnelle (cond, then , else) ->
      - code à produire : code de c, saut conditionnel vers label else (JUMPIF),
        code du then + saut àla fin , label du else, code du else, label de fin
      - utiliser getEtiquette() pour obtenir une nouvelle étiquette , distincte à chaque appel.
      - code_de_cond
        ^ "JUMPIF (0) " ^ lelse ^ "\n"
        ^ code_de_then
        ^ "JUMP " ^ lfinelse ^ "\n"
        ^ lelse ^ "\n"
        ^ code_de_else
        ^ lfinelse ^ "\n"
  | TantQue (c,b) ->
      - code à produire : label de début, code de c, saut conditionnel à la fin ,
        code de b, saut au début
  | Empty -> ""


and analyse_code_bloc li =
```

```
(* - déterminer la taille occupée par les variables locales de ce bloc
     (il peut être utile d'introduire une fonction auxiliaire qui donne
     la taille occupée par une instruction : Decl => taille du type / autre => 0)
   - générer le code pour la liste d'instructions
     suivi de la libération des variables locales (POP (0) taillevarloc) *)
  let taille = List.fold_right (fun i ti -> (taille_variables_declarees i) + ti) li 0 in
  let popfinal = "POP (0) " ^ (string_of_int taille) ^ "\n" in
  (analyse_code_li li) ^ popfinal

(* une liste d'instruction est un bloc dont on ignore la taille des variables locales *)
and analyse_code_li li =
  String.concat "" (List.map analyse_code_instruction li)

(* AstPlacement.fonction -> string *)
let analyse_code_fonction (Fonction(info, _, li, e)) =
  - Attention : li n'est pas un bloc (on peut utiliser les var locales dans e)
  - info doit être InfoFun(nom, typeRet, typeParams)
  - déterminer la taille des variables locales (comme ci-dessus)
  - déterminer la taille occupée par les paramètres (somme des tailles de typeParams)
  - générer le code avec :
    le nom (qui servira de label)
    + le code des instructions
    + le code de e
    + la libération des var locales : POP (taille typeret) taillevarloc
    + RETURN (taille typeret) tailleparam
  - explication des libérations (transparent 128) :
    - e laisse en sommet de pile le résultat de la fonction, et en dessous se trouvent
      les variables locales que l'on élimine via le POP
    - on a maintenant (de haut en bas) : la valeur de retour, l'enregistrement d'activation,
      les paramètres. RETURN va restaurer le compteur ordinal et libérer
      l'enregistrement et les paramètres.

Le code d'un Programme(fonctions,prog) est la concaténation du code de la
bibliothèque (getEntete) + code des fonctions + label "main" + code du bloc
prog + HALT.
```

Traduction des langages

Les pointeurs

Objectif :
- Etre capable d'ajouter les pointeurs au langage RAT
- Modifier le compilateur en conséquence

1 Les pointeurs

Une variable est physiquement identifiée de façon unique par son **adresse**, c'est-à-dire l'adresse de l'emplacement mémoire qui contient sa valeur.
Un **pointeur** est une variable qui contient l'**adresse** d'un autre objet informatique (une "variable de variable" en somme).

1.1 Déclaration

La déclaration d'un pointeur se fait selon la syntaxe suivante :

```
type* id;
```

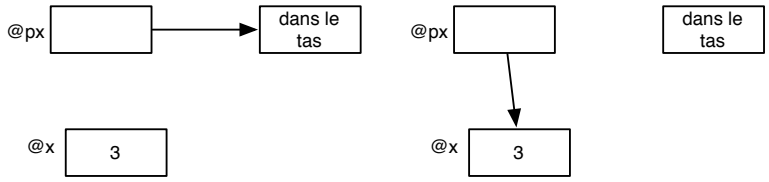
Cette instruction déclare une variable de nom id et de type pointeur(type) (pointeur sur une valeur de type type). Par exemple :

```
int* x;
```

déclare une variable x qui pointe sur une valeur de type int.

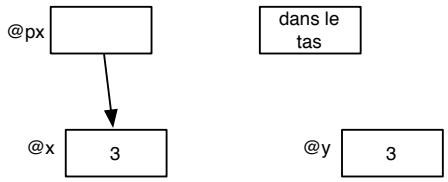
1.2 Adresse et valeur pointée

Les deux opérateurs particuliers en relation avec les pointeurs sont : & et *.
- & est l'opérateur qui **retourne l'adresse mémoire d'une variable**
  Si x est de type t alors &x est de type Pointeur(t)
  Exemple :
  ```
  int * px = (new int);
  int x = 3;
  px = &x;
  ```



- * est l'opérateur qui **retourne la valeur pointée par une variable pointeur**.
  Si x est de type Pointeur (t) alors *x est de type t
  Exemple :

```
int * px = (new int);
int x = 3;
px = &x;
int y = *px;
```



1.3 Exercice

▷ **Exercice 1** *Donner le type des variables dans le programme suivant :*

```
int * x = (new int);
int * * y = & x;
int z = 3;
*x = z;
*y = &z;
```

```
int * x = (new int);    // x.type = Pointeur(int)
int * * y = & x;      // y.type = Pointeur(Pointeur(int))
int z = 3;            // z.type=int
*x = z;              // (*x).type = z.type = int
*y = &z;             // (&z).type = Pointeur(int) = *y
```

▷ **Exercice 2** *Donner le type des variables dans le programme suivant :*

```
pointeur{
  int * x = (new int);
  * x = 3;
  int z= 18;
  int *y = & z;
  * y = *x;
}
```

```
pointeur{
  int * x = (new int);  // x.type = Pointeur(int)
  * x = 3;             // (*x).type = int
  int z= 18;           // z.type=int
  int *y = & z;        // y.type = Pointeur(int)
  * y = *x;
}
```

2 Modification de TAM et du compilateur

▷ **Exercice 3**

1. *Quelle(s) règles(s) faut-il modifier ou ajouter pour compléter l'introduction des pointeurs dans le langage ?*

   On simplifie les choses en limitant ce que l'on peut écrire :
   — $*$ **uniquement devant un identifiant mais en nombre quelconque** ($***x$)
   — & **unique et uniquement devant un identifiant (sens de** $\&\&z$ **?)**

   (a) $TYPE \rightarrow TYPE\ *$
   (b) $I \rightarrow A = E;$ **(remplace** $I \rightarrow id = E;$**)**
   (c) $E \rightarrow A$ **(remplace** $E \rightarrow id$**)**
   (d) $A \rightarrow *A$
   (e) $A \rightarrow id$
   (f) $E \rightarrow null$
   (g) $E \rightarrow (new\ TYPE)$ **(on récupère une nouvelle adresse dans le tas)**
   (h) $E \rightarrow \&\ id$

   Les parenthèses sont obligatoires autour de `new TYPE` pour que la grammaire reste analysable par Menhir. Cette forme de grammaire peut sembler compliquée, mais cela permet de préparer le projet.

2. *Quelles modifications faut-il apporter à l'AST ?*

   (a) Ajouter un cas de type : Pointeur **of** typ
   (b) Modifier un cas d'instruction : Affectation **of** affectable $*$ expression
   (c) Ajouter le nouveau type affectable = Ident **of** string | Valeur **of** affectable
   (d) Ajouter des cas aux expressions :
   — Affectable **of** affectable
   — Null
   — New **of** typ
   — Adresse **of** string

3. *Modifier la passe de gestion des identifiants.*

   Lors de la passe de gestion des identifiants, l'AST est modifié pour remplacer les identifiants par un pointeur sur leur information.
   Il faut ajouter à AstTds :

   (a) Modifier un cas d'instruction : Affectation **of** affectable $*$ expression
   (b)
   ```
   (* les affectables *)
   type affectable =
     | Ident of Tds.info_ast
     | Valeur of affectable
   ```
   (c) Ajouter des cas aux expressions :
   — Affectable **of** affectable
   — Null
   — New **of** typ
   — Adresse **of** Tds.info_ast

Actions à réaliser :

(a) **Ajout des méthodes d'analyse des affectables**

On notera que selon que l'affectable est en partie droite ou gauche d'une affectactation, le traitement n'est pas même : par exemple une constante est autorisée en partie droite mais pas en partie gauche. Il y a plusieurs possibilités pour résoudre ce problème, ici nous en proposons une avec un booléen (*modif*) qui indique si l'affectable est modifié (partie gauche d'une affectation) ou pas (les autres cas).

```
let rec analyse_tds_affectable tds (a:AstSyntax.affectable) modif =
  match a with
  | AstSyntax.Ident n ->
  begin
    match chercherGlobalement tds n with
    | None -> raise (IdentifiantNonDeclare n)
    | Some info ->
      begin
      match info_ast_to_info info with
      |InfoFun _ -> raise (MauvaiseUtilisationIdentifiant n)
      |InfoVar _ -> Ident info
      |InfoConst (_,v) -> if modif then raise (MauvaiseUtilisationIdentifiant n) else Ident info
      end
  end
  | AstSyntax.Valeur v -> Valeur (analyse_tds_affectable tds v)
```

(b) **Instruction :**

```
AstSyntax.Affectation (n,e) ->
  (* Dans affectation donc affectable en écriture *)
  Affectation ( analyse_tds_affectable tds n true, analyse_tds_expression tds e)
```

(c) **Expression :**

```
(* Dans expression donc affectable en lecture *)
  |AstSyntax.Affectable aff -> Affectable (analyse_tds_affectable tds aff false)
  |AstSyntax.Null -> Null
  |AstSyntax.New t -> New t
  |AstSyntax.Adresse n -> begin
      match chercherGlobalement tds n with
      | None -> raise (IdentifiantNonDeclare n)
      | Some info ->
        begin
        match info_ast_to_info info with
        |InfoFun _ -> raise (MauvaiseUtilisationIdentifiant n)
        |InfoVar _ -> Adresse info
        |InfoConst _ -> raise (MauvaiseUtilisationIdentifiant n)
        end
    end
```

4. *Modifier la passe de typage.*

Lors de la passe de typage les types sont supprimés de l'AST car ajoutés aux informations (pas de changement particulier pour les pointeurs) et l'AST est

modifié pour la résolution de surcharge (là encore pas de traitement particulier).

Il faut ajouter à AstTyp :

(a) **Modifier un cas d'instruction :** Affectation **of** affectable * expression

(b) **type** affectable = inchangé

(c) **Ajouter des cas aux expressions (inchangé) :**
- Affectable **of** affectable
- Null
- New **of** typ **(là on garde le type car pas d'info associé)**
- Adresse **of** Tds.info_ast

Actions à réaliser :

(a) **Dans `type.mli` (et `type.ml`) modifier le type** typ **pour ajouter un cas** Pointeur **of** typ**, et dans `type.ml` modifier** est_compatible **pour prendre en compte le cas pointeur.**

(b) **Ajout des méthodes d'analyse des affectables**

```
let rec analyse_type_affectable (aff:AstTds.affectable) : (affectable*typ) =
  match aff with
  | Ident info ->
  begin
    match info_ast_to_info info with
    | InfoVar (t,_,_) -> (Ident info, t)
    | InfoConst _ -> (Ident info, Int)
    | _ -> failwith ("Internal error: symbol not found")
  end
  | Valeur aff ->
  begin
    match analyse_type_affectable aff with
    | (naff, Pointeur t) -> Valeur (naff), t
    | _ -> raise NotAPointer
  end
```

(c) **Instruction :**

```
|AstTds.Affectation (n,e) ->
   let (naff,taff) = analyse_type_affectable aff in
   let (ne, texp) = analyse_type_expression e in
   if est_compatible texp taff
        then Affectation (naff, ne)
        else raise (TypeInattendu (texp,taff))
```

(d) **Expression :**

```
| AstTds.Affectable a -> let (na,t) = analyse_type_affectable a in (Affectable na,t)
| AstTds.Null -> Null, Pointeur Undefined
| AstTds.New t -> New t, Pointeur t
| AstTds.Adresse info ->     begin
    match info_ast_to_info info with
    | InfoVar (t,_,_) -> (Adresse info, Pointeur t)
    | _ -> failwith ("Internal error: symbol not found")
  end
```

5. *Modifier la passe de placement mémoire.*
   Rien à faire, il faut juste définir la taille du type pointeur (1).
6. *Proposer la traduction en TAM de l'exemple de l'exercice 2. On rappelle l'existence des instructions TAM suivantes, permettant de manipuler des adresses :*
   — *Empiler une adresse : LOADA d[r]*
   — *Empiler n mots à partir de l'adresse laissée en sommet de pile : LOADI (n)*
   — *Écrire n mots de la pile à l'adresse laissée en sommet de pile : STOREI (n)*
   — *Allocation de mémoire : SUBR MAlloc (réserve dans le tas une zone de la taille laissée en sommet de pile, l'adresse obtenue est laissée en sommet de pile).*

PUSH 1	**On réserve pour x**
LOADL 1	**Taille de l'entier**
SUBR MAlloc	**On réserve pour x**
STORE (1) 0[SB]	**On enregistre au déplacement de x, l'adresse ou sera stockée sa valeur**
LOADL 3	**On charge la constante**
LOAD (1) 0[SB]	**On charge l'adresse ou sera stockée la valeur de x**
STOREI (1)	**Ecrit la valeur de la constante à l'adresse de x**
PUSH 1	
LOADL 18	
STORE (1) 1[SB]	
PUSH 1	**La taille de y**
LOADA 1[SB]	**Empile l'adresse de z**
STORE (1) 2[SB]	**L'adresse de y reçois celle de z**
LOAD (1) 0[SB]	**x**
LOADI (1)	***x**
LOAD (1) 2[SB]	**y**
STOREI (1)	**Range**

On notera que

```
LOADL 18
STORE (2) 0[SB]
```

est une action identique à

```
LOADL 18
LOADA 0[SB]
STOREI (2)
```

Et que :

```
LOAD (2) 0[SB]
```

est une action identique à

```
LOADA 0[SB]
LOADI (2)
```

7. *Modifier la passe de génération de code.*
   Réflechir et coder !