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.

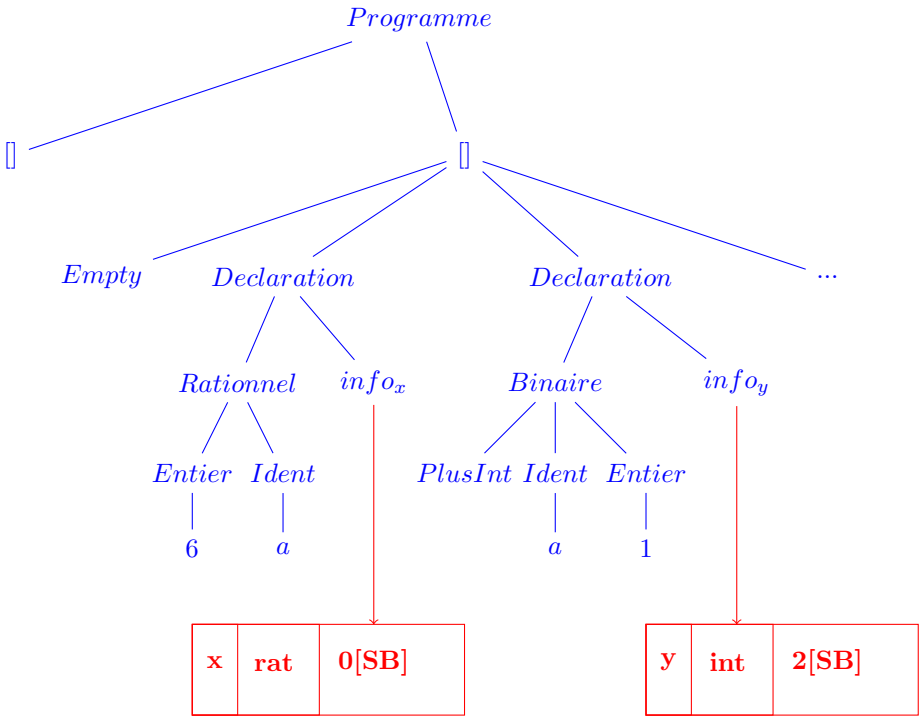
▷ **Exercice 2** *Définir les actions à réaliser lors de la passe de génération de code.*

On prend un extrait de l'exemple précédent :

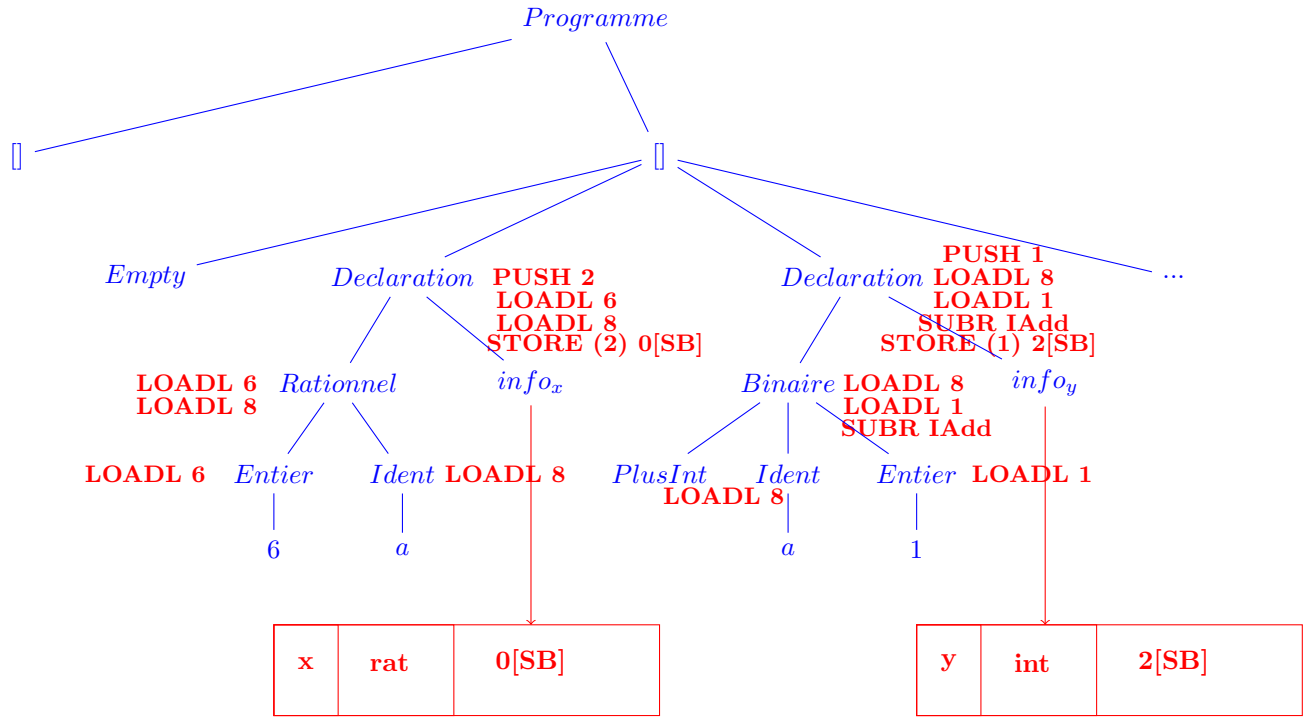
```
prog {
  const a = 8;
  rat x = [6/a];
  int y = (a+1);
 ...
}
```

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.
```