TD 5
TdL

Traduction des langages

Génération de code

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

1 De RAT à TAM

▷ **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);
  }
}
```

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.

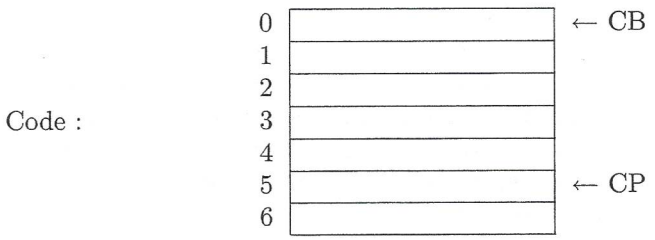
La dernière passe est une passe de génération de code. Il n'y a donc plus d'AST à générer.

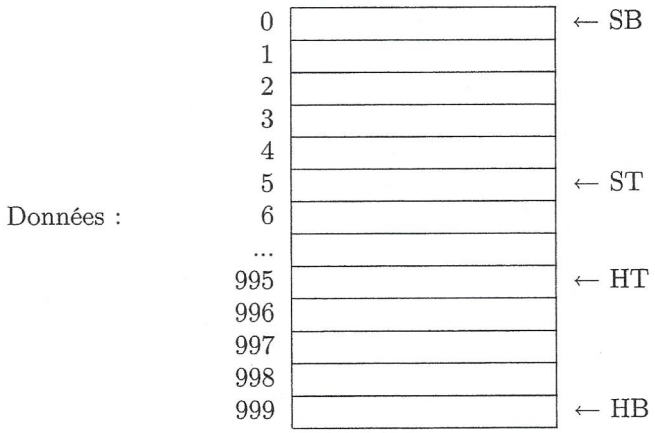
▷ **Exercice 2** *Définir les actions à réaliser lors de la passe de génération de code.*

TD 5.1
TdL

A La machine TAM

Machine à pile. Pas de registre de donnée.





Instructions (16) dont :

PUSH n	ST = ST+n
POP (d) n	a = ST -d; ST = ST - d -n; Pour i de d à 0 Donnees(ST++) = Donnees[a++] fin pour
LOADL n	Donnees(ST)= n; ST = ST+1
LOADA d[r]	ST = d[r] ; ST = ST + 1;
LOAD (n) d[r]	Pour i de 0 a n-1 Donnees(ST+i) = Donnees(val(r)+d+i) fin pour; ST = ST+n
STORE (n) d[r]	Pour i de 0 a n-1 Donnees(val(r)+d+i) = Donnees(ST+i-n); fin pour; ST = ST-n
JUMP etiq	CP = val(etiq)
JUMP d[r]	CP = val(r) + d
JUMPIF (n) etiq	si Donnees(ST -1) = n alors CP = val(etiq) fin si; ST = ST -1
JUMPIF (n) d[r]	si Donnees(ST -1) = n alors CP = val(r) + d fin si; ST = ST -1
LOADI (n)	Empile n mots lus à l'adresse précedemment empilée
STOREI (n)	Ecrit les n mots empilés, à l'adresse empilée
SUBR op	Appel de op, consommation des arguments laissés en sommet de pile
HALT	Arret

B Instructions de la machine TAM

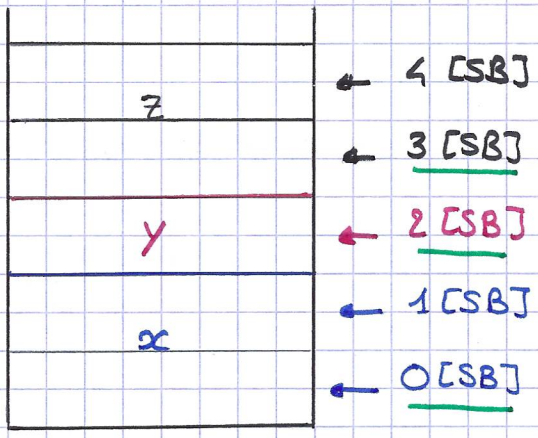
Nom	Paramètres	Résultat	
Fonctions Booléens			
BNeg	1	1	Négation logique
BOr	2	1	Ou logique
BAnd	2	1	Et logique
BOut	1	0	Affiche sur `stdout` un booléen (`true` ou `false`)
BIn	0	1	Lit sur `stdin` un booléen (`true` ou `false`)
B2C	0	1	Conversion vers un caractère (`true` = `'1'`, `false` = `'0'`)
B2I	0	1	Conversion vers un entier (`true` = `1`, `false` = `0`)
B2S	0	1	Conversion vers une chaîne (`"true"`, `"false"`)
Fonctions Caractères			
COut	1	0	Affiche sur `stdout` un caractère
CIn	0	1	Lit sur `stdin` un caractère
C2B	1	1	Conversion vers un booléen (`'1'` = `true`, `'0'` = `false`)
C2I	1	1	Conversion vers un entier (le code ASCII)
C2S	1	1	Conversion vers la chaîne contenant seulement ce caractère
Fonctions Entiers			
INeg	1	1	Négation entière
IAdd	2	1	Addition entière
ISub	2	1	Soustraction entière
IMul	2	1	Multiplication entière
IDiv	2	1	Diviseur dans division entière
IMod	2	1	Reste dans division entière
IEq	2	1	Test égalité entre 2 entiers
INeq	2	1	Test différence entre 2 entiers
ILss	2	1	Test inférieur strictement entre 2 entiers
ILeq	2	1	Test inférieur ou égal entre 2 entiers
IGtr	2	1	Test supérieur strictement entre 2 entiers
IGeq	2	1	Test supérieur ou égal entre 2 entiers
IOut	1	0	Affiche sur `stdout` un entier
IIn	0	1	Lit sur `stdin` un entier
I2B	1	1	Conversion vers un booléen (1 = `true`, 0 = `false`)
I2C	1	1	Conversion vers un caractère (le code ASCII)
I2S	1	1	Conversion vers la chaîne représentant cet entier
Fonctions Mémoires			
MVoid	0	1	Renvoie la valeur « adresse non initialisée »
MAlloc	1	1	Alloue un bloc mémoire et renvoie son adresse
MFree	1	0	Libère un bloc mémoire
MCompare	2	1	Test égalité entre le contenu de 2 blocs mémoire
MCopy	2	0	Copie le contenu d'un bloc mémoire dans le second bloc mémoire
Fonctions Chaînes			
SAlloc	1	1	Création d'une nouvelle chaîne
SCopy	1	1	Création d'une copie de la chaîne passée en paramètre
SConcat	2	1	Création d'une nouvelle chaîne contenant la juxtaposition de deux paramètres
SOut	1	0	Affiche sur `stdout` une chaîne
SIn	0	1	Lit sur `stdin` une chaîne
S2B	1	1	Conversion vers un booléen (`"true"` = `true`, `"false"` = `false`)
S2C	1	1	Extraction du premier caractère de la chaîne
S2I	1	1	Conversion vers l'entier représenté par la chaîne

TD5.2
TdL

Génération de code

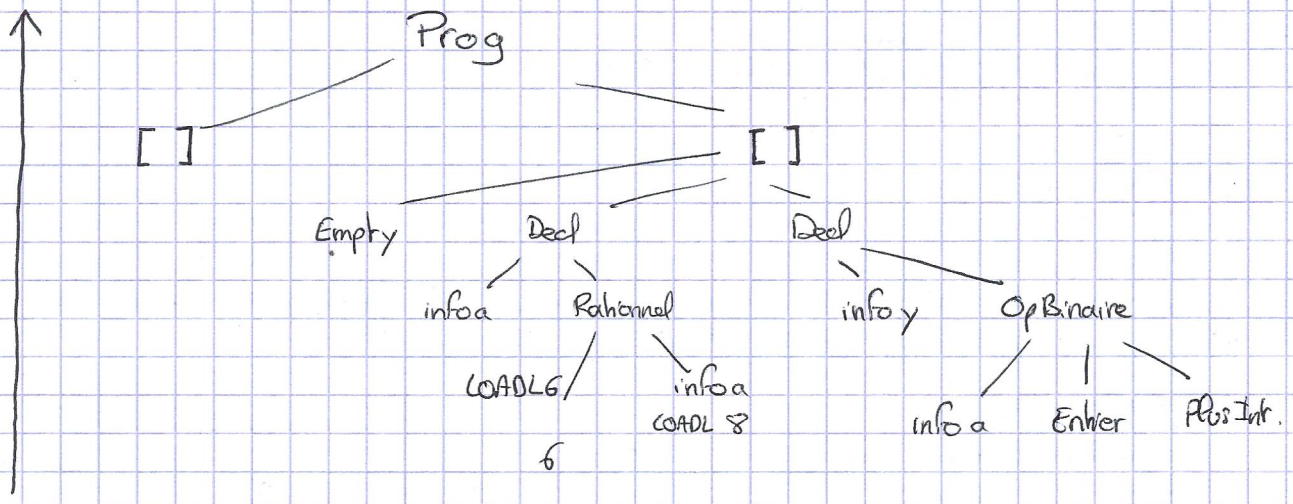
Exercice 1 :

Les constantes n'ont pas d'adresse, à chaque fois qu'on en rencontre une, elle est directement remplacée par sa valeur.

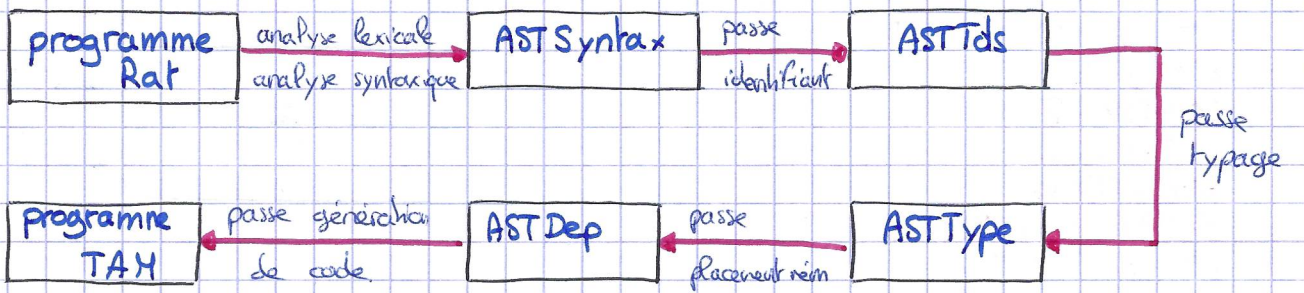



```
JUMP main
   (* fonctions *)  RAdd, RMul, ROut
main
PUSH 2
LOADL 6
LOADL 8
STORE (2) 0[SB]
PUSH 1
LOADL 8
LOADL 1
SUBR IAdd
STORE (1) 2[SB]
LOAD (2) 0[SB]
LOADL (3)
LOADL 2
CALL (-) RAdd
STORE (2) 0[SB]
debutwhile
  LOAD (1) 2[SB]
  LOADL 12
  SUBR ILss
  JUMPIF (0) finwhile
  PUSH 2
  LOAD (2) 0[SB]
  LOADL 5
  LOAD (1) 2[SB]
  CALL (-) RMul
  STORE (2) 3[SB]
  LOAD (2) 3[SB]
  CALL (-) ROut
  LOAD (1) 2[SB]
  LOADL 1
  SUBR IAdd
  STORE (1) 2[SB]
  POP (0) 2
  JUMP debutwhile
finwhile
  POP (0) 3
HALT
```

$x = x + [3/2]$



=> code en retour des fonctions d'analyse



Exercice 2 :

```
(* analyse_code_instruction : ASTDep -> string *)

let analyse_code_instruction i =
  match i with

  | Empty -> ""
  | Declaration (info, e) ->
      ("PUSH " ^ (taille type dans info) ")
        analyse_code_expression e
      ("STORE " ^ (taille) @ )
  | Affectation (info, e) ->
      analyse_code_expression e
      "STORE " ^ (taille) @,
  | Conditionnelle (e, bt, be) ->
      let etqbe = ...          in
      let etqfin = ...         in
        analyse_code_expression e
        + JUMPIF (0) etiq be
        + analyse_code_bloc bt
        + JUMP etiqfin
        + etiq be
        + analyse_code_bloc be
        + etiq fin
  | TantQue (e, b) ->
      let debutwhile = ...     in
      let fin while = ...      in
        debut while
        + analyse_code_expression e
        + JUMPIF (0) fin while
```

PUSH et STORE gobonnent si on retoie proprement la pile

TD 5.3
TdL

```
        + analyse_code_bloc b
        + JUMP debutwhile
        + finwhile
    | Affichage Int e ->
        analyse_code_expression e
        + SUBR Iout.
    | Affichage Bool e ->
        analyse_code_expression e
        + SUBR Bout
    | Affichage Rat e ->
        analyse_code_expression e
        + CALL (-) Rout.
```

```
(* analyse_code_bloc : ASTDep.bloc -> string *)

let analyse_code_bloc li =
    string.concat "" (List.map analyse_code_instruction li)
    + POP (0) taille_var_locales
```

↳ Fold_right (fun i ti -> (getTaille i) + ti) li 0