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.*

Module Tam. Pour rendre plus lisible la génération du code et pour éviter de bêtes erreurs de syntaxe TAM (oubli de passage à la ligne, parenthèses mal placées...), on fournit le module `Tam.mli` qui contient des fonctions de services pour générer chacune des instructions TAM :

⟶

```
(* LOAD (int) int[reg] *)
val load : int -> int -> string -> string

(* LOADA int[reg] *)
val loada : int -> string -> string

(* LOADI (int) *)
val loadi : int -> string

(* LOADL int *)
val loadl_int : int -> string

(* LOADL string *)
val loadl_string : string -> string

(* LOADL char *)
val loadl_char : char -> string

(* STORE (int) int[reg] *)
val store : int -> int -> string -> string

(* STOREI (int) *)
val storei : int -> string

(* CALL (reg) label *)
val call : string -> string -> string

(* RETURN (int) int *)
val return : int -> int -> string

(* SUBR label *)
val subr : string -> string

(* PUSH int *)
val push : int -> string

(* POP (int) int *)
val pop : int -> int -> string

(* JUMP label *)
val jump : string -> string

(* JUMPIF (int) label *)
val jumpif : int -> string -> string

(* HALT *)
val halt : string

(* label *)
val label : string -> string
```