3 Constructions `break` et `continue`

On considère un petit langage impératif avec variables globales, affectation, conditionnelle, boucle `while` et deux constructions `break` et `continue`. La syntaxe abstraite des instructions de ce langage est la suivante :

$$s ::= \begin{array}{l} x := e \\ | s ; s \\ | \text{if } e \text{ then } s \text{ else } s \\ | \text{while } e \text{ do } s \\ | \text{break} \\ | \text{continue} \\ | \text{skip} \end{array}$$

où e désigne une expression et x une variable. La sémantique des conditionnelles et des boucles est standard. L'instruction `skip` ne fait rien et `if  $e$  then  $s$  else skip` est une notation pour `if  $e$  then  $s$  else skip`. La sémantique des constructions `break` et `continue` est la suivante : `break` sort immédiatement de la boucle englobante (la plus interne), et `continue` reprend immédiatement au test de la boucle englobante (la plus interne).

Question 1 Donner la valeur de la variable x à la fin de l'exécution de l'instruction suivante :

```
x := 0; i := 0;
while true do
  if x > 20 then break;
  i := i + 1;
  if i mod 2 = 0 then continue;
  x := x + i
```

Question 2 Donner un type OCaml `stmt` pour la syntaxe abstraite des instructions, en supposant un type `expr` pour la syntaxe abstraite des expressions et le type `string` pour représenter les variables.

Question 3 On impose deux conditions sur les instructions `break` et `continue` :

1. toute instruction `break` ou `continue` doit apparaître à l'intérieur d'une boucle `while`;
2. dans toute séquence $s_1; s_2$, l'évaluation de s_1 ne doit pas systématiquement atteindre une instruction `break` ou `continue` (ce qui rendrait s_2 inatteignable).

Écrire une fonction OCaml `check : stmt → bool` vérifiant ces deux conditions (en supposant qu'on ne peut rien affirmer quant aux expressions).

Question 4 Écrire une fonction OCaml `compile_stmt : stmt → asm` pour compiler une instruction vers du code assembleur, en supposant un type `asm` pour le code assembleur et une fonction `compile_expr : expr → asm` pour compiler une expression vers du code assembleur qui stocke sa valeur dans le registre `%rax`. On supposera que toute variable x est stockée en mémoire à une adresse désignée par l'étiquette x .