CCMP 2023 MPI Informatique II corrigé

☐ 1 – La chaîne x_0 contient un symbole 1 et 10^{10} symboles 0, donc $K(y_0) \leqslant 10^{10} + 1$.

☐ 2 – Si n est pair, alors $n = 2n'$, donc $10^n = (10^{n'})^2$. Si est est impair, alors $n = 2n' + 1$, donc $10^n = (10^{n'})^2 \times 10$. On en déduit le code suivant :

```
let rec exp10 n =
  if n = 0 then 1
  else
    let n' = n / 2 in
    let e = exp10 n' in
    if n mod 2 = 0 then e * e
    else e * e * 10
in string_of_int (exp10 (exp10 10));;
```

Ce code contient moins de 200 caractères, donc $K(y_0) \leqslant 200$.

☐ 3 – Les entiers étant bornés, l'exécution du code précédent provoquerait des dépassements de capacité et le résultat ne serait pas correct. De plus, sur une machine ayant peu de mémoire, la fonction `exp10` n'étant pas récursive terminale, on pourrait également dépasser la capacité de la pile d'exécution.

☐ 4 – C'est une question bien plus difficile qu'il n'y paraît. Utiliser l'écriture de n en base 256 semble naturel, mais cela provoque des erreurs si l'on ne fait pas attention : 0 peut être associé à la chaîne vide ou au caractère de code 0, ou bien on peut « oublier » toutes les chaînes de caractères ayant un nombre quelconque de caractères de code 0 à gauche, correspondant aux chiffres nuls de poids fort que l'on négligerait.

On propose de numéroter les écritures en base 256 **de taille fixe**, en les rangeant par tailles et valeurs croissantes : d'abord l'écriture vide, puis les écritures de taille 1 (0, 1, …, 255), puis les écritures de taille 2 (0 0, 0 1, …, 255 255), etc. Il suffira ensuite, pour un entier n, de prendre l'écriture en base 256 de numéro n et de remplacer les chiffres de cette écriture par les caractères associés dans la numérotation ASCII. On obtient bien une bijection : la numérotation est injective par construction, et surjective car toute chaîne de caractère peut être interprétée comme une représentation en base 256 de même taille.

Pour implémenter cette transformation, on commence par écrire une fonction auxiliaire qui prend un tableau d'entiers représentant l'une de ces écritures en base 256 et qui renvoie l'écriture suivante dans la numérotation. Pour cela, on essaie d'incrémenter l'entier représenté par le tableau et, en cas de dépassement de capacité, on renvoie la représentation de 0 avec un chiffre de plus.

```
let increment (t : int array) : int array =
  let n = Array.length t in
  let i = ref (n - 1) in
  while !i >= 0 && t.(!i) = 255 do
    t.(!i) <- 0;
    decr i
  done;
  if !i >= 0 then (t.(!i) <- t.(!i) + 1; t)
  else Array.make (n + 1) 0
```

Une fois cette fonction écrite, on peut déterminer la représentation associée à l'entier n par incrémentations successives, puis renvoyer la chaîne de caractères associée.

```
let phi (n : int) : string =
  let rec loop (n : int) : int array =
    match n with
    | 0 -> [||]
    | 1 -> [|0|]
    | _ -> increment (loop (n - 1))
  in
  let t = loop n in
  String.init (Array.length t) (fun i -> Char.chr t.(i))
```

☐ 5 – On exploite une recherche dichotomique, en cherchant tout d'abord une borne supérieure sur la valeur de m.

```
let psi (m : int) : int =
  let ub = ref 1 in
  while (kolmogoroff (phi !ub)) < m do
    ub := 2 * !ub
  done;
  let lb = ref 0 in
  while !lb < !ub do
    let k = (!lb + !ub) / 2 in
    if (kolmogoroff (phi k)) >= m then ub := k
    else lb := k + 1
  done;
  !ub
```

☐ 6 – $K(\varphi(\psi(m))) \geqslant m$ par définition de $\psi(m)$. Pour représenter $\varphi(\psi(m))$, on peut utiliser le code des fonctions `phi` et `psi` des questions précédentes (de taille constante par rapport à la valeur de m), que l'on appelle sur l'entier m représenté sous forme de chaîne de caractères à l'aide de $\mathcal{O}(\log_{10} m)$ caractères. On a donc un code représentant $\varphi(\psi(m))$ de taille $\mathcal{O}(\log m)$, donc par définition de la complexité de Kolmogoroff $K(\varphi(\psi(m))) = \mathcal{O}(\log m)$.

Comme $K(\varphi(\psi(m))) = \mathcal{O}(\log m)$, lorsque m tend vers $+\infty$, $\frac{K(\varphi(\psi(m)))}{m}$ tend vers 0. Pourtant, $\frac{K(\varphi(\psi(m)))}{m} \geqslant 1$ d'après l'inégalité démontrée en début de question. La fonction `kolmogoroff` ne peut donc pas exister : K n'est pas une fonction calculable.

☐ 7 – C'est un compilateur.

☐ 8 – Pour obtenir une description de y, on peut prendre une description de y par rapport à $\mathcal{D}$, et lui appliquer le décompresseur `d`. Cela nous donne un code de taille $K_{\mathcal{D}}(y)$, plus la taille du code de `d`, plus quelques caractères (comme des espaces) pour la mise en forme du code et de l'appel. Ces caractères supplémentaires et le code de `d` ne dépendant pas de y, on peut compter tout cela dans la constante $c_{\mathcal{D}}$. Par définition de la complexité de Kolmogoroff, on a donc bien $K(y) \leqslant K_{\mathcal{D}}(y) + c_{\mathcal{D}}$.

☐ 9 – Le code commence par la déclaration d'une fonction récursive `t` d'entrée un entier `e` (entier car on le compare à `0`), et de sortie un entier également (on renvoie `1` dans une branche). Le code de `t` se termine au niveau du second **in**. Il suit une déclaration d'une variable `n` valant `t 10`, puis le calcul de `t n`. La chaîne x_0 dénote donc une expression entière, donc de type `int`.

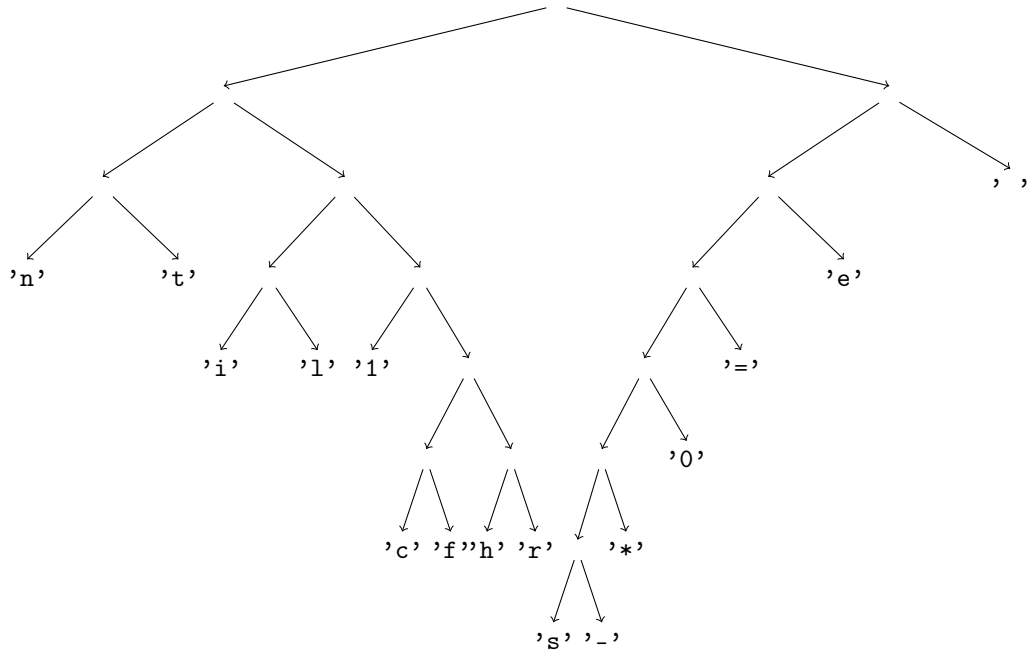
☐ 10 – La fonction récursive `t` est la fonction $e \mapsto 10^e$ (cela se prouve par récurrence). L'évaluation de x_0 renvoie donc la valeur $10^{(10^{10})}$.

☐ 11 – Ce code est peu lisible en raison de sa mise en forme : pas de retours à la ligne, pas d'indentations, pas de noms de fonctions/variables significatifs.

☐ 12 – L'idée est d'initialiser un dictionnaire vide et, pour chaque caractère de la chaîne, d'incrémenter l'entier qui lui est associé s'il existe et d'ajouter l'association avec l'entier 1 s'il n'existe pas d'association pour ce caractère. Il est à noter que la fonction `Hashtbl.add` ne fait pas ce que l'on veut ici car elle ne remplace pas les associations existantes : elle en crée de nouvelles. On utilise `Hashtbl.replace` à la place.

```
let count (x : string) : (char, int) Hashtbl.t =
  let d = Hashtbl.create (String.length x) in
  for i = 0 to String.length x - 1 do
    let c = x.[i] in
    match Hashtbl.find_opt d c with
    | None -> Hashtbl.add d c 1
    | Some n -> Hashtbl.replace d c (n + 1)
  done;
  d
```

☐ 13 – On applique directement l'algorithme de construction d'une arbre de Huffman en suivant la recommandation de l'énoncé.



☐ 14 – On commence par construire la table qui à chaque caractère associe sa représentation compressée, ainsi que la taille de cette représentation :

'n'	't'	'i'	'l'	'1'	'c'	'f'	'h'	'r'	's'	'-'	'*'	'0'	'='	'e'	' '
000	001	0100	0101	0110	011100	011101	011110	011111	1000000	1000001	100001	10001	1001	101	11
3	3	4	4	4	6	6	6	6	7	7	6	5	4	3	2

On peut alors multiplier ces tailles par les nombres d'occurrences des caractères (grâce au dictionnaire obtenu via la fonction `count`) et sommer les résultats. Cela donne : 239.

On a donc une description de y_0 par rapport à $\mathcal{H}$ de taille 239. Par définition, on sait donc que $K_{\mathcal{H}}(y_0) \leqslant 239$.

☐ 15 – La portée de `seed1` est limitée au code de `new_string1`. Celle de `seed2` est limitée au code de `new_string2 ()`, i.e. cette variable n'existe que dans un appel de la fonction `new_string2` et est redéfinie à chaque appel. La variable `seed3` est de portée globale.

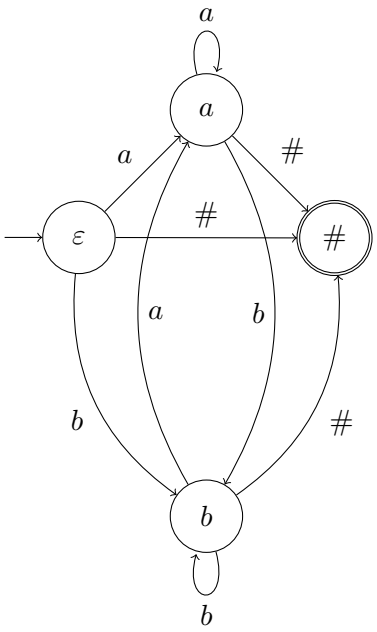
☐ 16 – D'après ce qui précède, la fonction `new_string2` est la fonction constante égale à `"a#"` , donc ne respecte pas la spécification. Pour la distinguer des autres, il suffit d'écrire l'assertion suivante, qu'elle est la seule à ne pas passer :

```
assert (let _ = new_string () in let s = new_string () in s = "
    aa#")
```

☐ 17 – La fonction `new_string3` peut poser problème parce que la portée globale de `seed3` laisse la porte ouverte à des modifications de cette variable en dehors de la fonction. On pourrait donc réinitialiser sa valeur et produire des chaînes qui ne sont pas inédites avec la fonction `new_string3`.

☐ 18 – Avec une dérivation, on peut produire un nombre quelconque de a et de b, suivi du symbole $\#$. On a donc $L(\mathcal{G}_0) = (a \mid b)^*\#$.

☐ 19 – En appliquant l'algorithme de Berry-Sethi, on obtient :



☐ 20 – On lit les caractères tant qu'on lit des `'a'` et des `'b'` et on s'arrête au premier `'#'` s'il existe.

```
let rec parseV (w : char list) : string * char list =
  match w with
  | '#' :: s -> "#", s
  | c :: q when c = 'a' || c = 'b' ->
     let v, s = parseV q in
     String.make 1 c ^ v, s
  | _ -> raise SyntaxError
```

☐ 21 – On montre par induction sur une dérivation de $w \in L(\mathcal{G}_1)$ que tout préfixe strict non vide de w a strictement moins de parenthèses fermantes que de parenthèses ouvrantes, et que w a exactement autant de parenthèses fermantes que de parenthèses ouvrantes. On procède par disjonction de cas selon la première dérivation immédiate :

- $T \Rightarrow \_$: pas de parenthèses, ni de préfixes stricts non vides, le résultat est vrai ;
- $T \Rightarrow (TT)$: il existe donc $u, v \in L(G_1)$ tels que $w = (uv)$. On observe alors les préfixes stricts non vides de w :
  - (: le résultat est vrai ;
  - (u', avec u' préfixe strict non vide de u : par hypothèse d'induction, u' a strictement moins de parenthèses fermantes que de parenthèses ouvrantes, donc (u' aussi ;
  - (u : par hypothèse d'induction, u a exactement autant de parenthèses fermantes que de parenthèses ouvrantes, donc (u a exactement une parenthèse fermante de moins que de parenthèses ouvrantes ;
  - (uv', avec v' préfixe strict non vide de v : on conclut par le point précédent et l'hypothèse d'induction ;
  - (uv : on conclut par le point sur (u et l'hypothèse d'induction.

  L'hypothèse sur u et v assure également que w a exactement autant de parenthèses fermantes que de parenthèses ouvrantes.

Ainsi, si $w' \in L(\mathcal{G}_1)$ et si w est un préfixe strict non vide de w', alors on ne peut avoir $w \in L(\mathcal{G}_1)$ car w devrait avoir exactement autant de parenthèses fermantes que de parenthèses ouvrantes et strictement moins de parenthèses fermantes que de parenthèses ouvrantes à la fois.

☐ 22 – On rappelle que l'ambiguïté s'exprime aussi par l'unicité de la dérivation à gauche de chaque mot du langage. Supposons que $w \in L(\mathcal{G}_1)$ admet deux dérivations à gauche distinctes et soit de taille minimale vis-à-vis de cette propriété. Alors $w \neq \_$ car $\_$ n'admet qu'une seule dérivation, donc a fortiori une seule dérivation à gauche. Ainsi, w admet deux dérivations à gauche distinctes, toutes deux commençant par la dérivation immédiate $T \Rightarrow (TT)$. Il existe donc deux décompositions $w = (uv) = (u'v')$ avec $u, v, u', v' \in L(\mathcal{G}_1)$.

Comme $L(\mathcal{G}_1)$ est sans préfixe, u n'est pas préfixe de u' et u' n'est pas préfixe de u, donc la seule possibilité est que $u = u'$, donc $v = v'$. Mais dans ce cas, u ou v admet deux dérivations à gauche distinctes, ce qui est impossible par minimalité de w.

☐ 23 – On procède comme à la question précédente, le symbole `var` remplaçant le symbole $\_$. La preuve de la question précédente montre que notre contre-exemple minimal ne peut être de la forme `var` ou (uv). Il reste donc la forme `var->`T, qui se traite simplement en exploitant la minimalité de w.

☐ 24 – Il y a deux cas dans l'écriture du code :

- on lit une parenthèse ouvrante, et cela ne peut être qu'une application ;
- on lit une variable, et comme on cherche un préfixe maximal on vérifie si c'est le début du code d'une fonction avant de se contenter de la variable.

```
let rec parseT (w : char list) : ada * char list =
  match w with
  | '(' :: q ->
     let (a1, q1) = parseT q in
     let (a2, q2) = parseT q1 in
     begin
       match q2 with
       | ')' :: q' -> A (a1, a2), q'
       | _ -> raise SyntaxError
     end
  | _ ->
     let (v, q') = parseV w in
     match q' with
     | '-' :: '>' :: w' ->
        let a, s = parseT w' in
        F (v, a), s
     | _ -> V v, q'
```

☐ 25 – On fait appel à `parseT` et on vérifie que tous les caractères ont été lus.

```
let parse (w : string) : ada =
  match parseT (charlist_of_string w) with
  | a, [] -> a
  | _ -> raise SyntaxError
```

☐ 26 – Il n'existe pas d'automate reconnaissant $\{\lceil n \rceil \mid n \in \mathbb{N}\}$: si c'était le cas, en notant n le nombre d'états de cet automate, on pourrait appliquer le lemme de l'étoile à $\lceil n \rceil$, et obtenir une décomposition $\lceil n \rceil = xyz$ avec $|xy| \leqslant n$, $y \neq \varepsilon$ et $xy^*z \subseteq \{\lceil n \rceil \mid n \in \mathbb{N}\}$. Comme $|xy| \leqslant n$ et $y \neq \varepsilon$, la suppression de y pour obtenir xz va :

- endommager le préfixe `b#->a#->` ;
- ou endommager un facteur `b#` entre deux parenthèses ouvrantes ;
- ou modifier le nombre de parenthèses ouvrantes sans modifier les parenthèses fermantes.

Le mot xz n'appartient donc pas au langage $\{\lceil n \rceil \mid n \in \mathbb{N}\}$, ce qui est absurde.

☐ 27 – On vérifie d'abord que le préfixe est le bon, afin d'identifier les variables, puis on compte le nombre d'applications en vérifiant que les variables sont les bonnes.

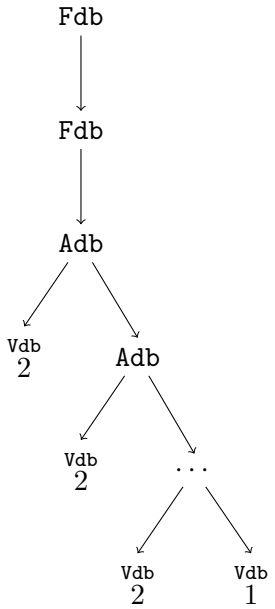
```
let int_of_ada (a : ada) : int =
  match a with
  | F (v2, F (v1, a)) ->
     let rec aux (a : ada) : int =
       match a with
       | V v when v = v1 -> 0
       | A (V v, a') when v = v2 -> 1 + aux a'
       | _ -> raise SyntaxError
     in aux a
  | _ -> raise SyntaxError
```

☐ 28 – Une structure d'arbre binaire de recherche équilibré, comme les arbres bicolores, semble indiquée.

☐ 29 – On applique directement la définition.

```
let rec free_vars (a : ada) : StringSet.t =
  match a with
  | V v -> StringSet.singleton v
  | A (a1, a2) -> StringSet.union (free_vars a1) (free_vars a2)
  | F (v, a) -> StringSet.remove v (free_vars a)
```

☐ 30 – Dans $\lceil n \rceil$, `a` sera numéroté 1 et `b` sera numéroté 2, d'où l'arbre :



☐ 31 – On utilise une fonction récursive auxiliaire pour garder en compte le nombre de constructeurs `F` rencontrés. On suppose ici que le terme de De Bruijn est clos.

```
let ada_of_tdb (t : tdb) : ada =
  let rec aux (t : tdb) (n : int) : ada =
    match t with
    | Vdb k -> V (string_of_int k)
    | Adb (t1, t2) -> A (aux t1 n, aux t2 n)
    | Fdb t -> F (string_of_int (n + 1), aux t (n + 1))
  in aux t 0
```

☐ 32 – Le code $\hat{t}$ commence par 0000 en raison des deux symboles `Fbd`. Ensuite, pour chaque `Abd` (il y en a n), on a les caractères 01 suivis de 110 (le code de `Vbd 2`). Enfin, on termine par le code de `Vbd 1`, à savoir 10. On a donc $\left|\hat{t}\right| = 5n + 6$.

☐ 33 – Supposons qu'un code w soit commun à deux termes distincts t, t' et soit de taille minimale vis-à-vis de cette propriété.

- Si w commence par 1, alors $w = \underbrace{1 \dots 1}_{u \text{ fois}} 0$ et t et t' valent tous deux `Vdb u`, ce qui contredit $t \neq t'$.
- Si w commence par 00 ($w = 00w'$), alors t et t' sont de la forme `Fdb s` et `Fdb s'` avec $s \neq s'$ de code w'. La minimalité de w est donc contredite.
- Si w commence par 01, alors $w = 01\hat{t_1}\hat{t_2}$ avec t s'écrivant `Adb (`t_1`,` t_2`)`, et $w = 01\hat{t'_1}\hat{t'_2}$ avec t' s'écrivant `Adb (`t'_1`,` t'_2`)`. Deux cas :

- $t_1 = t'_1$: dans ce cas $\hat{t_1} = \hat{t'_1}$ et $\hat{t_2} = \hat{t'_2}$ avec $t_2 \neq t'_2$ contredisant la minimalité de w.
- $t_1 \neq t'_1$: si $\hat{t_1} = \hat{t'_1}$, alors cela contredit la minimalité de w, sinon on suppose sans perte de généralité que $\left|\hat{t_1}\right| < \left|\hat{t'_1}\right|$. On a alors un préfixe strict de $\hat{t'_1}$ qui est un code binaire d'un terme de De Bruijn. Ceci est impossible car le codage binaire est sans préfixe.

Pour montrer que le codage binaire est sans préfixe, on considère par l'absurde un code minimal w' tel qu'il existe un préfixe strict w de w' qui soit également un codage binaire de terme de De Bruijn. On observe alors la forme des codes :

- si le code commence par 1, alors c'est le code d'un terme `Vdb u` et il n'y a pas d'autre façon d'avoir un code bien formé que d'écrire $\underbrace{1\ldots1}_{u \text{ fois}}0$, donc $w = w'$, ce qui est absurde ;
- si le code commence par 00, alors il est de la forme $00\hat{t}$ avec t un terme. On aurait donc $w = 00u$ et $w' = 00u'$ avec u, u' des codages binaires et u préfixe strict de u', ce qui contredit la minimalité de w' ;
- si le code commence par 01, alors il est de la forme $01\hat{t_1}\hat{t_2}$, donc on aurait $w = 01uv$ et $w' = 01u'v'$, avec u, u', v, v' des codages binaires. Si $|u| < |u|'$, u est préfixe strict de u', ce qui contredit la minimalité de w'. De même si $|u|' < |u|$, cela contredit la minimalité de w'. Donc $u = u'$, et comme w est préfixe strict de w', v est préfixe strict de v', ce qui contredit la minimalité de w'.

☐ 34 – On lit les caractères un à un à l'aide d'une fonction auxiliaire récursive qui reconnaît un préfixe correspondant à un terme de De Bruijn et qui renvoie ce terme et l'indice du dernier caractère du code de ce terme.

```
let decode (z : string) : tdb =
  let n = String.length z in
  let rec read_tdb (i : int) : tdb * int =
    if i = n then raise SyntaxError
    else if z.[i] = '1' then
      (* cas Vdb *)
      let j = ref i in
      while !j < n && z.[!j] = '1' do
        incr j
      done;
      if !j = n then raise SyntaxError
      else Vdb (!j - i), !j
    else if i = n - 1 then raise SyntaxError
    else if z.[i + 1] = '0' then
      (* cas Fdb *)
      let t, j = read_tdb (i + 2) in
      Fdb t, j
    else
      (* cas Adb *)
      let t1, j1 = read_tdb (i + 2) in
      let t2, j2 = read_tdb (j1 + 1) in
      Adb (t1, t2), j2
  in
```

```
let t, j = read_tdb 0 in
if j <> n-1 then raise SyntaxError
else t
```

☐ 35 – On applique directement la définition inductive.

```
let rec substitute (v : string) (by_a : ada) (in_b : ada) : ada =
  match in_b with
  | V v1 -> if v = v1 then by_a else in_b
  | A (b1, b2) -> A (substitute v by_a b1, substitute v by_a b2)
  | F (v1, b1) ->
     if v = v1 then in_b
     else if not (StringSet.mem v1 (free_vars by_a)) then
       F (v1, substitute v by_a b1)
     else
       let v1' = new_string () in
       F (v1', substitute v by_a (substitute v1 (V v1') b1))
```

☐ 36 – On applique directement la définition inductive.

```
let rec reduce_one_step (a : ada) : ada =
  match a with
  | V _ -> raise NoReduction
  | A (a1, a2) ->
     begin match a1 with
     | V _ -> A (a1, reduce_one_step a2)
     | F (v, a11) -> substitute v a2 a11
     | _ -> A (reduce_one_step a1, a2)
     end
  | F (v, a1) -> F (v, reduce_one_step a1)
```

☐ 37 – On appelle récursivement la fonction d'interprétation sur la réduction en un pas de son argument, en rattrapant l'exception indiquant qu'il n'y a plus de réduction possible.

```
let rec interpret (a : ada) : ada =
  try interpret (reduce_one_step a) with
  | NoReduction -> a
```

☐ 38 – y_0 correspondant à $10^{10^{10}}$, on veut construire un mot z_0 tel que `ada_of_tdb (decode z0)` est égal à `parse pi`, où le mot π est défini comme dans l'énoncé avec $n = 10$. On a vu en question ☐ 32 – comment construire le code t_n du mot $\lceil n \rceil$ à l'aide de $5n+6$ symboles. Le code z_0 recherché s'écrit donc $01000101101010t_{10}$. Ce code est de taille $14+5\times10+6 = 70$. On a donc une description de y_0 par rapport à $\mathcal{B}$ de taille 70, et par définition de la complexité de Kolmogoroff par rapport à $\mathcal{B}$, on sait que $K_{\mathcal{B}}(y_0) \leqslant 70$.