Composition d'informatique n°5

Corrigé

⋆ ⋆ ⋆

1 Complexité de Kolmogoroff

Question 1 La chaîne v_0 est constituée d'un 1 suivi de 10^{10} zéros. On en déduit que $K(v_0) \leqslant 1 + 10^{10}$.

Question 2 On a :
- $10^n = (10^m)^2$ si n est pair ;
- $10^n = (10^m)^2 \times 10$ sinon.

On en déduit la fonction `exp10` pour calculer 10^n et le code suivant :

```
let rec exp10 = function
    | 0 -> 1
    | n ->
        let x = exp10 (n / 2) in
        if n mod 2 = 0 then x * x
        else x * x * 10
in
string_of_int (exp10 (exp10 10))
```

On en déduit donc que $K(v_0) \leqslant 200$.

Question 3 On peut mentionner les problèmes suivants (certains n'en sont en fait pas) :
- les entiers étant représentés sur un nombre fini fixe de bits, usuellement 63 ou 31 en OCaml, le calcul de $10^{10^{10}}$ mènerait nécessairement à un dépassement d'entiers, donc des résultats incohérents (voire des erreurs si on essaie de créer une chaîne de taille négative) ;
- la fonction étant récursive, on peut se soucier du dépassement de la taille de la pile d'appels. Toutefois, le nombre d'appels récursifs est de l'ordre de $\log_2(10^{10}) = 10 \log_2 10 \leqslant 40$;
- on peut s'inquiéter du temps de calcul, mais celui-ci reste humainement accessible, car il y a de l'ordre de 40 multiplications d'entiers bornés.

En remarque : si on veut travailler avec un type spécial d'entiers non bornés, c'est le temps de calcul lié à la multiplication qui devient problématique.

Question 4 On peut se baser sur une représentation en base 256, mais il faut apporter quelques modifications pour obtenir la bijectivité (pour s'assurer que 0 est envoyé sur le mot vide, par exemple). On propose la transformation suivante :
- 0 est encodé en la chaîne vide ε ;
- un entier $n > 0$ a pour premier caractère celui de code ASCII $(n-1) \mod 256$, et pour suffixe l'encodage de $\left\lfloor \frac{n-1}{256} \right\rfloor$.

On obtient alors :

```
let rec phi = function
    | 0 -> ""
    | n ->
        let c = Char.chr ((n - 1) mod 256) in
        String.make 1 c ^ phi ((n - 1) / 256)
```

Question 5 On écrit une simple boucle pour faire le calcul.

```
let psi m =
    let n = ref 0 in
    while kolmogorof (phi !n) < m do
        incr n
    done;
    !n
```

Question 6 Par définition, $K(\varphi(\psi(m))) \geqslant m$, car $\psi(m) \in \{n \in \mathbb{N} \mid K(\varphi(n)) \geqslant m\}$ (c'est son minimum). De plus, si on considère le code suivant :

```
let kolmogorov = ...  (* code de la fonction *)
let rec phi n = ...
let psi m = ...

let m = ...  (* écriture de m en base 10 *)
in
phi (psi m)
```

alors ce code est une description de $\varphi(\psi(m))$. Sa taille est donc un majorant de $K(\varphi(\psi(m)))$. Mais l'écriture des trois fonctions nécessite un nombre constant fixe de caractères. De plus, l'écriture de l'entier m en base 10 nécessite un nombre logarithmique en m de chiffres. On en déduit que $K(\varphi(\psi(m))) = \mathcal{O}(\log m)$.

Cela est contradictoire avec le fait que $K(\varphi(\psi(m))) \geqslant m$. On en déduit que la fonction `kolmogorov` ne peut pas exister.

Question 7 Cela s'appelle un transpilateur, mais on peut voir cela comme un cas particulier de compilateur.

Question 8 De manière similaire à précédemment, si on considère le code :

```
let d z = ...  (* code source de d *)

let z = ...  (* description de y par rapport à D *)

eval (d z)
```

Alors on obtient un programme qui a pour valeur de retour y. Par ailleurs, si on considère $c_{\mathcal{D}}$ comme étant le nombre de caractères de ce code source autre que la description de y par rapport à $\mathcal{D}$, alors pour une telle description minimale, ce code source est bien de longueur $K_{\mathcal{D}}(y) + c_{\mathcal{D}}$, d'où la borne voulue.

2 Estimation de la complexité grâce au décompresseur de Huffman

Question 9 La fonction `t` prend en argument `e`. Comme il y a un test `e=0`, `e` est de type `int`. De plus, le retour `then 1` montre que le type de retour est aussi `int`.

Finalement, après la définition de la fonction, on trouve `let n = t 10 in t n`, donc le type de retour est `int`.

Question 10 La fonction `t` est une fonction qui à `e` associe 10^e. On en déduit que la valeur de l'expression est $10^{10^{10}}$, c'est-à-dire l'entier décrit par y_0 (donné en introduction).

Question 11 On peut citer plusieurs problèmes :

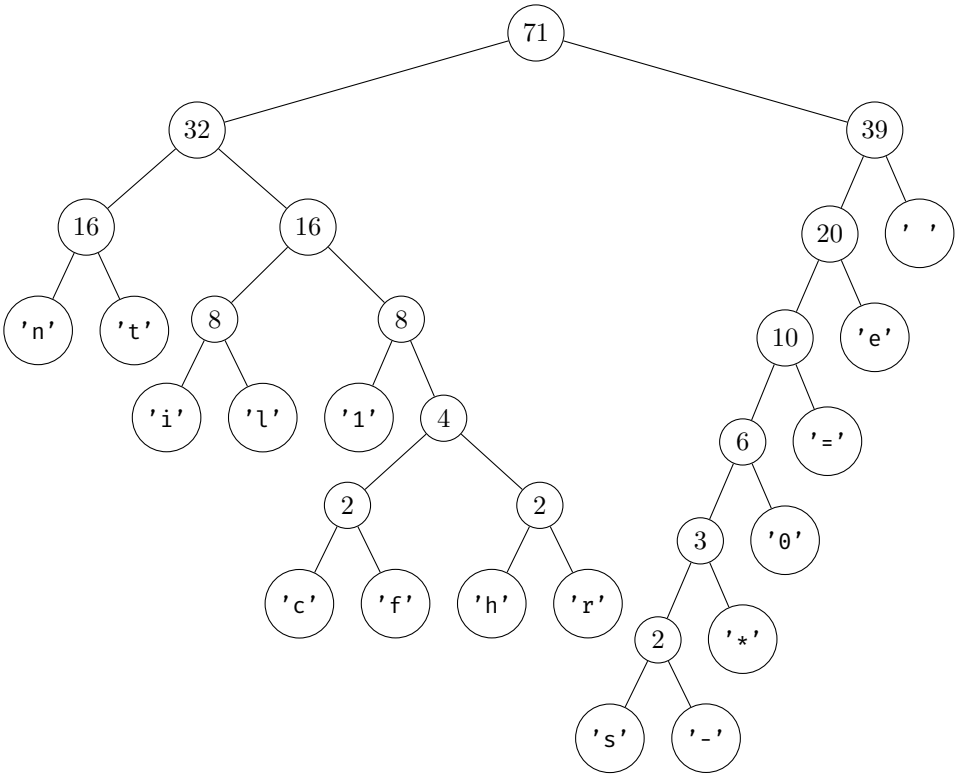
– il n'y a pas de sauts de lignes ou d'espaces autour des opérateurs, permettant d'améliorer la lisibilité ;

– le nom de la fonction `t` n'est pas très explicite ;
– les noms de variables ne sont pas très cohérents (`e`, puis `n`, pour désigner dans les deux cas un exposant).

Question 12 On parcourt toute la chaîne de caractères, et on ajoute une entrée à la table pour chaque nouvelle lettre, en ajoutant 1 au nombre actuel d'occurrences.

```
let count x =
    let h = Hashtbl.create 1 in
    let n = String.length x in
    for i = 0 to n - 1 do
        let c = x.[i] in
        let occ = 1 + match Hashtbl.find_opt h c with
            | None -> 0
            | Some k -> k
        in
        Hashtbl.add h c occ
    done;
    h
```

Question 13 On suit l'indication sur la position des feuilles. On indique, dans chaque nœud interne, le nombre d'occurrences d'une des lettres dans le sous-arbre.



Question 14 Il suffit ici de calculer la somme $\sum_{\sigma \in \Sigma} |x|_\sigma \times p_\sigma$, où p_σ est la profondeur du caractère σ dans l'arbre précédent.

On peut compléter le tableau de l'énoncé avec les profondeurs :

lettre	'n'	't'	'i'	'l'	'1'	'c'	'f'	'h'	'r'	's'	'-'	'*'	'0'	'='	'e'	' '
occurrences	8	8	4	4	4	1	1	1	1	1	1	1	3	4	10	19
profondeur	3	3	4	4	4	6	6	6	6	7	7	6	5	4	3	2

On obtient une chaîne de longueur 239, qui est la borne demandée par l'énoncé (la chaîne z_0 est une description de y_0 par rapport à $\mathcal{H}$).

3 Interlude

Question 15 La variable `seed1` est locale, mais existe encore après la définition de `new_string1`. La variable `seed2` est locale à la fonction `new_string2`. La variable `seed3` est globale.

Question 16 La proposition 2 ne respecte pas la spécification, car elle génèrera toujours la même chaîne de caractères.

Comme test, on peut par exemple écrire :

```
let _ = assert (new_string2 () <> new_string2 ())
```

qui échouera.

Question 17 La solution 3 peut engendrer des erreurs de manipulation, car la variable `seed3` est globale et pourrait être modifiée par erreur. Dans la solution 1, la variable `seed1` existe, mais n'est plus accessible, donc ne peut pas être modifiée autrement que par un appel à `new_string1`.

4 Estimation de la complexité grâce au décompresseur de De Bruijn

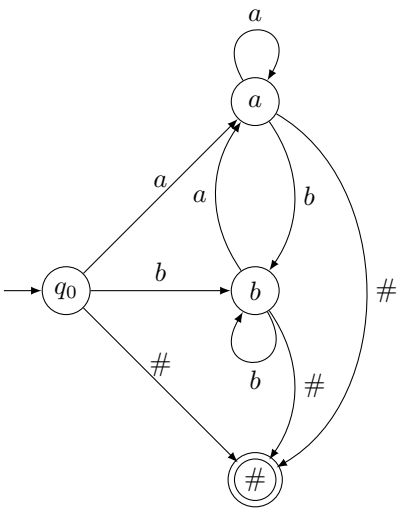
4.1 Construction syntaxique d'un langage

Question 18 C'est l'ensemble des mots de $\{a, b\}^*$ terminés par un #. On peut donner l'expression régulière $(a \mid b)^*\#$.

Question 19 L'expression régulière e précédente est déjà linéaire. On calcule les différents ensembles :

- $P(e) = \{a, b, \#\}$;
- $S(e) = \{\#\}$;
- $F(e) = \{aa, ab, a\#, ba, bb, b\#\}$.

Comme le mot vide n'est pas dans le langage, on obtient l'automate :



Question 20 On fait bien attention à respecter les types demandés (`char list` ou `string`). L'idée est de s'arrêter si le mot commence par # ou s'il est vide. Sinon, on parse la fin du mot, et on concatène la première lettre.

```
let rec parseV = function
    | '#' :: s -> "#", s
    | 'a' :: w' -> let v', s = parseV w' in "a" ^ v', s
    | 'b' :: w' -> let v', s = parseV w' in "b" ^ v', s
    | _ -> raise SyntaxError
```

Question 21 Montrons par récurrence forte sur la taille d'une dérivation les deux propriétés suivantes sur un mot $w \in L(\mathcal{G}_1)$:

- tout préfixe strict de w contient strictement plus de parenthèses ouvrantes que fermantes, sauf si $w = \texttt{\_}$;
- w contient autant de parenthèses ouvrantes et fermantes.

En effet :

- si la dérivation est de taille 1, c'est-à-dire $T \Rightarrow \texttt{\_}$, alors les deux propriétés sont bien vérifiées ;
- sinon, la dérivation de w est de la forme $T \Rightarrow \texttt{(}TT\texttt{)} \Rightarrow^* \texttt{(}w_1w_2\texttt{)} = w$, avec $T \Rightarrow^* w_1$ et $T \Rightarrow^* w_2$. Par hypothèse de récurrence, w_1 et w_2 vérifie la propriété, donc on en déduit que w aussi (en distinguant selon la longueur d'un préfixe strict).

Dès lors, si $w, w' \in L(\mathcal{G}_1)$ sont tels que $w' = ws$ avec $s \neq \varepsilon$, alors comme $|w|_{\texttt{(}} = |w|_{\texttt{)}}$, on en déduit que w est un préfixe strict de w' avec autant de parenthèses ouvrantes que fermantes, donc $w' = \texttt{\_}$, ce qui est absurde (car $|w|' \geqslant 2$).

Question 22 Supposons $\mathcal{G}_1$ ambiguë et soit w le plus petit mot ambigu pour $\mathcal{G}_1$.

Alors comme $w \neq \texttt{\_}$ (qui ne possède qu'une seule dérivation), ces dérivations s'écrivent $T \Rightarrow_g \texttt{(}TT\texttt{)} \Rightarrow_g^* \texttt{(}w_1w_2\texttt{)}$, avec $T \Rightarrow_g^* w_1$ et $T \Rightarrow_g^* w_2$, et $T \Rightarrow_g \texttt{(}TT\texttt{)} \Rightarrow_g^* \texttt{(}w_1'w_2'\texttt{)}$, avec $T \Rightarrow_g^* w_1'$ et $T \Rightarrow_g^* w_2'$.

De plus, $w_1 \neq w_1'$, car sinon w_1 ou w_2 serait ambigu, ce qui contredirait la minimalité de w. On en déduit que w_1 est préfixe de w_1' ou l'inverse, ce qui contredit la question précédente.

Par l'absurde, on en déduit que $\mathcal{G}_1$ n'est pas ambiguë.

Question 23 On montre qu'il y a à nouveau unicité de dérivations gauches, de manière un peu plus informelle. Lorsqu'on dérive une variable T, on distingue selon les deux premières lettres du mot à obtenir :

- si la première est `(`, la seule règle possible est $T \to \texttt{(}TT\texttt{)}$;
- si les deux premières sont `var` et `->`, la seule règle possible est $T \to \texttt{var->}T$. En effet, si on utilise la règle $T \to \texttt{var}$, il ne sera pas possible de faire apparaître `->` ;
- sinon, la seule règle possible est $T \to \texttt{var}$.

Question 24 On distingue les cas en fonction de la première lettre de w. Si le mot commence par `(`, on essaie de lire deux fois de suite un mot du langage, on vérifie que les deux mots sont suivis par une parenthèse fermante, puis on renvoie l'arbre ainsi formé et la suite. Sinon, on essaie de lire une variable (avec la fonction `parseV`), et on distingue selon que cette variable soit suivie par `->` ou non.

```
let rec parseT = function
    | [] -> raise SyntaxError
    | '(' :: w' ->
        let a1, s1 = parseT w' in
        let a2, s2 = parseT s1 in
        begin match s2 with
            | ')' :: s -> A (a1, a2), s
            | _ -> raise SyntaxError
        end
    | w ->
        begin match parseV w with
            | v, '-' :: '>' :: w' ->
                let a, s = parseT w' in
                F (v, a), s
            | v, s -> V v, s
        end
```

Question 25 À noter, la fonction donnée en indication peut s'écrire simplement, de la forme :

```
let charlist_of_string s =
    List.init (String.length s) (fun i -> s.[i])
```

Pour écrire la fonction demandée, on se sert de la fonction `parseT`, qu'on appelle sur w, et on vérifie que le suffixe restant est vide.

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

Question 26 On note $L = \{\lceil n \rceil \mid n \in \mathbb{N}\}$. Montrons que le langage L n'est pas rationnel, en utilisant le lemme de pompage. Supposons qu'il soit rationnel, et soit n sa longueur de pompage. On pose $u = \lceil n \rceil$ et $u = xyz$ sa décomposition donnée par le lemme de pompage. On distingue les cas :

- si y contient un `a`, `-` ou `>`, alors xy^2z contient au moins trois de ces caractères, donc n'est pas dans L ;
- si $|x| < 8$, alors xy^2z contient plus que 8 lettres avant le dernier `>`, donc n'est pas dans le langage.

On en déduit que y est un facteur du mot `(b#(b#... (b#` (car $|xy| \leqslant n$). À nouveau, on distingue :

- si y contient `(`, alors xz ne contient pas autant de parenthèses ouvrantes que fermantes, donc n'est pas dans le langage ;
- sinon, y est un facteur non vide compris entre deux parenthèses ouvrantes consécutives, donc est égal à `b`, `#` ou `b#`. On en déduit que xy^2z contient un facteur `bb`, `##` ou `b#b#`, donc n'est pas un mot du langage.

Dans tous les cas, on arrive à une contradiction, donc le langage n'est pas rationnel, donc pas reconnu par un automate fini.

Question 27 On s'arrête sur l'incarnation de 0. Sinon, on vérifie que le mot commence bien comme une incarnation, et on enlève un niveau d'arbre avant de relancer un appel récursif.

```
let rec int_of_ada = function
    | F (v2, F (v1, V v1')) when v1 = v1' -> 0
    | F (v2, F (v1, F (v2', a))) when v2 = v2' -> 1 + int_of_ada (F(v2, F(v1, a)))
    | _ -> raise SyntaxError
```

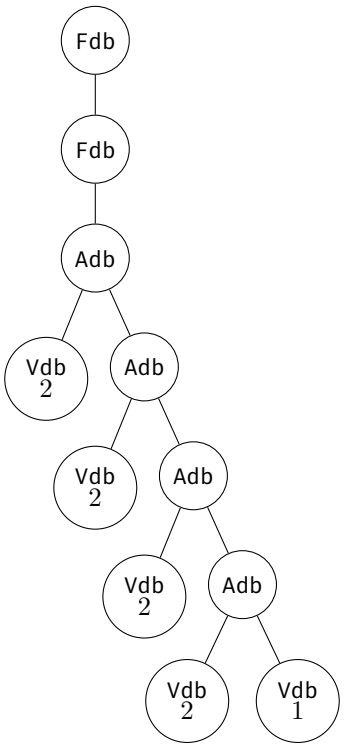
4.2 Réécriture des variables et sérialisation

Question 28 Une table de hachage permet d'implémenter un ensemble avec des opérations usuelles en temps constant en moyenne.

Question 29 On se contente de suivre la définition inductive qui nous est donnée.

```
let rec free_vars = function
    | V v -> StringSet.singleton v
    | A (a1, a2) -> StringSet.union (free_vars a1) (free_vars a2)
    | F (v, a1) -> StringSet.remove v (free_vars a1)
```

Question 30 On obtient l'arbre suivant (ici pour $\lceil 4 \rceil$, qu'on généralise) :



Question 31 Pour chaque nœud `Fdb`, on crée une nouvelle variable, avec la fonction `new_string1`. Il est sûrement possible de réutiliser des variables entre différents sous-arbres, mais cette méthode a l'avantage d'être plus simple.

La fonction récursive auxiliaire prend en argument la liste des variables liées présentes dans le sous-arbre. Lorsqu'on tombe sur un nœud `Vdb`, on cherche alors la bonne variable liée, en utilisant le numéro. On utilise à cette occassion la fonction `List.nth` (dont l'utilisation pertinente est plutôt rare...).

```
let ada_of_tdb t =
    let rec ada_rec vars = function
        | Vdb i -> V (List.nth vars (i - 1))
        | Fdb t ->
            let v = new_string1 () in
            F (v, ada_rec (v :: vars) t)
        | Adb (t1, t2) -> A (ada_rec vars t1, ada_rec vars t2)
    in
    ada_rec [] t
```

Question 32 Soit t le terme associé à $\lceil n \rceil$. Alors d'après la question 30, $\hat{t} = 0000(01110)^n10$. On a donc $|\hat{t}| = 5n + 6$.

Question 33 Supposons qu'il existe $b = b_1 \ldots b_k \in \{0,1\}^*$ et deux termes de De Bruijn $t \neq t'$ tels que $\hat{t} = \hat{t'}$. Supposons de plus b de taille minimale vérifiant cette propriété. On distingue :

- si $b_1 = 1$, alors par construction, $t = t' = \texttt{Vdb}(k-1)$, ce qui est absurde ;
- si $b_1 b_2 = 00$, alors par construction, $t = \texttt{Fdb}t_1$, $t' = \texttt{Fdb}t'_1$, avec $\hat{t_1} = \hat{t'_1} = b_3 \ldots b_k$. Par minimalité de b, on en déduit que $t_1 = t'_1$, donc $t = t'$, ce qui est absurde ;
- si $b_1 b_2 = 01$, alors $t = \texttt{Adb}(t_1, t_2)$ et $t' = \texttt{Adb}(t'_1, t'_2)$, avec $\hat{t_1}\hat{t_2} = \hat{t'_1}\hat{t'_2} = b_3 \ldots b_k$. On distingue à nouveau :
  - si $\hat{t_1} = \hat{t'_1}$, alors $\hat{t_2} = \hat{t'_2}$, donc par hypothèse d'induction, $t_1 = t'_1$ et $t_2 = t'_2$, puis $t = t'$, ce qui est absurde ;
  - sinon, sans perte de généralité, $\hat{t_1}$ est un préfixe strict de $\hat{t'_1}$. C'est absurde, car le codage binaire est sans préfixe (voir ci-après).

Dans tous les cas, on arrive à une contradiction, ce qui permet de conclure sur l'injectivité.

Justifions maintenant que le codage binaire est sans préfixe : supposons qu'il existe u, u' des codages binaires, avec u préfixe strict de u'. Supposons de plus que $u' = b_1 \ldots b_k$ est de taille minimale. Distinguons :

- si $b_1 = 1$, alors comme précédemment, $u' = u$, ce qui est absurde ;
- si $b_1 b_2 = 00$, alors $u = 00v$ et $u' = 00v'$, avec v et v' des codages binaires. v est donc un préfixe strict de v', ce qui contredit la minimalité de u' ;
- si $b_1 b_2 = 01$, alors $u = 01vw$ et $u' = 01v'w'$. Alors v est préfixe strict de v' ou v' est préfixe strict de v. Dans les deux cas, cela contredit la minimalité de u'.

Question 34 On raisonne comme on l'a fait précédemment : on écrit une fonction récursive qui prend en argument un mot z (sous forme de liste de charactères), et renvoie le terme de De Bruijn correspondant au plus long préfixe de z étant un codage binaire, ainsi que le suffixe privé de ce codage. À partir de cette fonction, on décode une chaîne en appliquant la fonction précédente, et en vérifiant que le reste est vide.

```
let decode z =
    let rec decode_rec = function
        | '1' :: '0' :: s -> Vdb 1, s
        | '1' :: u ->
            begin match decode_rec u with
                | Vdb n, s -> Vdb (n + 1), s
                | _ -> raise SyntaxError
            end
        | '0' :: '0' :: u ->
            let t, s = decode_rec u in
            Fdb t, s
        | '0' :: '1' :: u ->
            let t1, s1 = decode_rec u in
            let t2, s2 = decode_rec s1 in
            Adb (t1, t2), s2
        | _ -> raise SyntaxError
    in
    match decode_rec (charlist_of_string z) with
        | t, [] -> t
        | _ -> raise SyntaxError
```

4.3 Interpréteur et décompresseur de De Bruijn

Question 35 On suppose ici que l'énoncé veut dire $[v_1 \leftarrow \texttt{V}v'_1]$ au lieu de $[v_1 \leftarrow v'_1]$. On applique la définition inductive donnée.

```
let rec substitute v by_a in_b = 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_string1 () in
            let b1' = substitute v by_a (substitute v1 (V v1') b1) in
            F (v1', b1')
```

Question 36 À nouveau, on se contente d'appliquer la réduction.

```
let rec reduce_one_step = function
    | A (a1, a2) ->
        begin match a1 with
            | V 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)
    | _ -> raise NoReduction
```

Question 37 On applique ce qui est demandé, en gérant bien la levée d'exception.

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

Question 38 On considère π tel que défini dans l'énoncé, avec $n = 10$. On pose alors z_0 le codage binaire du terme de De Bruijn associé à π. On a donc $z_0 = 01000101101010\widehat{\lceil 10 \rceil}$. Ce mot est de taille $14 + 5 \times 10 + 6 = 70$.

Par construction, `ada_of_tdb (decode z0)` est égal à `parse pi`, dont l'interprétation est une incarnation de $10^{10^{10}}$. On en déduit que `decompB z0` renvoie bien la chaîne y_0, donc que $K_{\mathcal{B}}(y_0) \leqslant |z_0| = 70$.

⋆ ⋆ ⋆