A2023 – INFO MPI II

Concours commun

Mines-Ponts

ÉCOLE DES PONTS PARISTECH,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.

Concours Mines-Télécom

CONCOURS 2023

DEUXIÈME ÉPREUVE D'INFORMATIQUE

Durée de l'épreuve : 4 heures

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :

INFORMATIQUE II - MPI

L'énoncé de cette épreuve comporte 12 pages de texte.

Cette épreuve concerne uniquement les candidats de la filière MPI.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Préliminaires

L'épreuve est composée d'un problème unique, comportant 38 questions. Le problème est divisé en quatre sections qui peuvent être traitées séparément, à condition de lire toutes les définitions de la section 1. Dans la première section (page 1), nous introduisons la *complexité de Kolmogoroff* et étudions des propriétés de calculabilité. Dans la deuxième section (page 3), nous estimons la complexité de Kolmogoroff à l'aide du codage de Huffman. La troisième section (page 4) contient des prolégomènes pour la section suivante. Dans la quatrième section (page 5), nous nous appuyons sur un modèle de calcul épuré, introduit à travers plusieurs langages formels, afin toujours d'estimer la complexité de Kolmogoroff.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractère différentes désigne la même entité, mais du point de vue mathématique pour la police en italique (par exemple n, $\mathcal{D}$ ou π) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`, `d` ou `pi`).

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Il faudra coder des fonctions à l'aide du langage de programmation OCaml exclusivement, en reprenant l'en-tête de fonctions fourni par le sujet, sans s'obliger à recopier la déclaration des types. Il est permis d'utiliser la totalité du langage OCaml mais il est recommandé de s'en tenir aux fonctions les plus courantes afin de rester compréhensible. Des rappels ponctuels de documentation du langage OCaml peuvent être proposés à titre d'aide. Quand l'énoncé demande de coder une fonction, sauf demande explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1 Complexité de Kolmogoroff

Nous notons Σ l'ensemble ordonné des 256 caractères ASCII étendus usuels et Σ^* l'ensemble des chaînes de caractères. Pour toute chaîne de caractères $x \in \Sigma^*$, la longueur de x, notée $|x|$, est le nombre de caractères qui la composent. Par exemple, la longueur de la chaîne `"abac"` est 4. Le nombre d'occurrences d'un symbole $\sigma \in \Sigma$ dans une chaîne de caractères $x \in \Sigma^*$ est noté $|x|_\sigma$. Par exemple, $|\texttt{abac}|_{\texttt{a}} = 2$.

Dans l'ensemble du sujet, nous nous appuyons sur une *machine universelle*, c'est-à-dire une fonction OCaml `eval`, de type `string -> string`, telle que :

— si la chaîne x, de type `string`, est le code source d'une expression OCaml y de type `string` et si l'exécution du code x se termine sans erreur, alors `eval x` se termine et a pour valeur de retour la valeur de y ;

— sinon, `eval x` ne se termine pas.

Les exécutions de `eval` ont lieu sur une machine idéale dont la mémoire est infinie et qui est capable de gérer des types natifs de taille quelconque.

Définition : Pour toute chaîne de caractères $y \in \Sigma^*$, nous disons que la chaîne de caractères $x \in \Sigma^*$ est une *description* de la chaîne y si le calcul `eval x` se termine et renvoie la chaîne y. Nous appelons *complexité de Kolmogoroff* de y et notons $K(y)$ la longueur de la plus courte chaîne de caractères $x \in \Sigma^*$ qui décrit y.

L'objet de ce sujet est d'étudier des propriétés et diverses majorations de la complexité de Kolmogoroff. Dans toutes nos illustrations, nous nous concentrons sur la description de la chaîne de caractères

$$y_0 = \texttt{"1000000..."} \in \Sigma^*,$$

qui correspond à l'entier $10^{(10^{10})}$ écrit en base 10 et que nous fixons une fois pour toutes.

1.1 Un premier exemple

☐ 1 – Proposer une première majoration de la complexité de Kolmogoroff $K(y_0)$, qui s'appuie sur la chaîne de caractères $x_0 = y_0 = \texttt{"1000000..."}$ comme description de y_0.

Indication OCaml : Il est rappelé que la fonction `string_of_int` convertit un entier en une chaîne de caractères.

☐ 2 – Soient n un entier naturel et n' la partie entière de $\frac{n}{2}$. Exprimer la quantité 10^n en fonction de $10^{n'}$. Compléter le code OCaml suivant en utilisant une stratégie « diviser pour régner » :

```
let exp10 n = (* Calcul de 10^n a ecrire *)
in string_of_int (exp10 (exp10 10))
```

afin d'en faire une description de la chaîne de caractères $y_0 = \texttt{"1000000..."}$. En déduire une nouvelle borne grossière (à 10^2 près) de la complexité de Kolmogoroff $K(y_0)$, significativement meilleure que celle de la question 1.

☐ 3 – Décrire quelle ou quelles difficultés adviendraient si l'on exécutait le code de la question 2 sur une machine réelle.

1.2 Quelques propriétés

Indication OCaml : L'expression `String.make (n : int) (sigma : char) : string` désigne la chaîne de caractères répétant n fois le caractère $\sigma \in \Sigma$. Pour tout entier n compris entre 0 et 255, `Char.chr (n : int) : char` désigne le n^{e} caractère dans la numérotation ASCII.

☐ 4 – Présenter une bijection $\varphi : \mathbb{N} \to \Sigma^*$ entre l'ensemble des entiers naturels et l'ensemble de chaînes de caractères. En écrire le code sous la forme d'une fonction OCaml `phi (n : int) : string`.

Nous prétendons avoir écrit une fonction OCaml `kolmogoroff (y : string) : int` qui calcule la complexité de Kolmogoroff $K(y)$.

☐ 5 – Écrire une fonction OCaml `psi (m : int) : int` dont la valeur de retour est l'entier

$$\psi(m) = \min\{n \in \mathbb{N}\,;\; K(\varphi(n)) \geq m\}$$

où $\varphi : \mathbb{N} \to \Sigma^*$ est la bijection définie à la question 4 et qui utilise la fonction `kolmogoroff`.

☐ 6 – Établir d'une part que, pour tout entier naturel m, on a

$$K(\varphi(\psi(m))) \geq m$$

et d'autre part que l'on a

$$K(\varphi(\psi(m))) = O(\log m).$$

Discuter l'existence de la fonction OCaml `kolmogoroff`.

Définition : Nous appelons *décompresseur* toute fonction OCaml `d : string -> string`. Pour toute chaîne de caractères $y \in \Sigma^*$ et pour tout décompresseur $\mathcal{D}$ (noté informatiquement `d`), nous disons que la chaîne de caractères z est une *description* de la chaîne y par rapport à $\mathcal{D}$ si le calcul `eval (d z)` se termine sans erreur et a pour valeur de retour y. Nous appelons *complexité de Kolmogoroff par rapport à* $\mathcal{D}$ de la chaîne y, et notons $K_{\mathcal{D}}(y)$, la longueur de la plus courte chaîne de caractères $z \in \Sigma^*$ qui décrit y par rapport à $\mathcal{D}$.

☐ 7 – Dire comment se nomme en informatique un programme qui transforme un code source dans un certain langage de programmation en un code équivalent dans un second langage.

☐ 8 – Montrer que pour tout décompresseur $\mathcal{D}$, il existe une constante entière $c_{\mathcal{D}}$ telle que, pour toute chaîne de caractères y, nous avons :

$$K(y) \leq K_{\mathcal{D}}(y) + c_{\mathcal{D}}.$$

2 Estimation de la complexité grâce au décompresseur de Huffman

Nous fixons dans cette section la chaîne de caractères $x_0 \in \Sigma^*$ suivante

```
"let rec t e=if e=0 then 1 else let n=e-1 in 10*t n in let n=t 10 in t n".
```

La chaîne x_0 contient 71 caractères, l'espace étant un caractère et les guillemets ne faisant pas partie de la chaîne.

☐ 9 – Inférer le type OCaml de l'expression dénotée par la chaîne de caractères x_0.

☐ 10 – Déterminer la valeur de l'évaluation de `x0` en tant que code source OCaml.

☐ 11 – Signaler une ou plusieurs caractéristiques du code source x_0 qui rend la lecture de ce code impénétrable par un humain.

Indication OCaml : Nous rappelons le détail de quelques fonctions du module OCaml `Hashtbl` permettant de manipuler des dictionnaires (ou tableaux associatifs) mutables.

— `Hashtbl.create (n : int) : ('a, 'b) Hashtbl.t` crée un dictionnaire vide de taille initiale n.
— `Hashtbl.add (d : ('a, 'b) Hashtbl.t) (k : 'a) (v : 'b) : unit` ajoute une association entre la clé k et la valeur v au dictionnaire d.
— `Hashtbl.find_opt (d : ('a, 'b) Hashtbl.t) (k : 'a) : 'b option` vaut `Some v` si le dictionnaire `d` possède une association entre la clé k et la valeur v et vaut `None` sinon.

☐ 12 – Écrire une fonction OCaml `count (x : string) : (char, int) Hashtbl.t` dont la valeur de retour est un dictionnaire qui associe chaque caractère $\sigma \in \Sigma$ présent dans la chaîne x à son nombre d'occurrences $|x|_\sigma$.

Nous exécutons `count x0` et obtenons le dictionnaire suivant :

'n'	't'	'i'	'l'	'1'	'c'	'f'	'h'	'r'	's'	'-'	'*'	'0'	'='	'e'	' '
8	8	4	4	4	1	1	1	1	1	1	1	3	4	10	19

Nous appelons z_0 la chaîne de bits correspondant à la compression de la chaîne x_0 par l'algorithme de Huffman.

☐ 13 – Dessiner un arbre de Huffman associé à la chaîne de caractères x_0. Il est recommandé de placer les feuilles de gauche à droite comme dans le tableau ci-dessus.

Nous notons $\mathcal{H}$ le décompresseur qui utilise l'arbre de Huffman de la question 13 pour transformer une chaîne de bits $z \in \{0, 1\}^*$ en un mot $x \in \Sigma^*$ et renvoie la chaîne de caractères "`string_of_int` (x)".

☐ 14 – Calculer la longueur $|z_0|$ de la chaîne obtenue après compression de x_0. En déduire que la complexité de Kolmogoroff de $y_0 = \texttt{10000}\ldots$ par rapport au décompresseur $\mathcal{H}$ vérifie

$$K_{\mathcal{H}}(y_0) \leq 239.$$

3 Interlude

Nous souhaitons écrire une fonction `new_string : unit -> string` ainsi spécifiée : à chaque appel, une chaîne de caractères inédite est produite. Nous offrons trois propositions de code.

```
let inc s = s := "a" ^ (!s); !s    (* Commun aux 3 propositions *)

let new_string1 =                  (* Proposition 1 *)
  let seed1 = ref "#" in
  fun () -> inc seed1

let new_string2 () =               (* Proposition 2 *)
  let seed2 = ref "#" in
  inc seed2

let seed3 = ref "#"                (* Proposition 3 *)
let new_string3 () =
  inc seed3
```

☐ 15 – Analyser la portée de la variable `seed1`, respectivement `seed2` et `seed3`, dans la fonction `new_string1`, respectivement `new_string2` et `new_string3`.

☐ 16 – Déduire de la question 15 laquelle des trois fonctions `new_string1`, `new_string2` et `new_string3` ne respecte pas la spécification. Écrire un test qui permet de discriminer la fonction erronée.

☐ 17 – Déduire de la question 15 laquelle des deux fonctions correctes restantes est plus propice à des erreurs de manipulation. Expliquer.

4 Estimation de la complexité grâce au décompresseur de De Bruijn

Nous nous avisons que la complexité de Kolmogoroff est influencée par la verbosité et l'expressivité du langage de programmation. Dans cette section, nous tentons de réduire l'encombrement ou la facilité d'écriture dus à la syntaxe du langage OCaml en introduisant un modèle de calcul nouveau et épuré.

4.1 Construction syntaxique d'un langage

Nous présentons systématiquement les *grammaires* sous la forme d'un quadruplet (N, Γ, S, Π) où N désigne l'alphabet des symboles non terminaux, Γ l'alphabet des symboles terminaux, $S \in N$ le symbole initial et Π l'ensemble des règles de production, de la forme $X \to \gamma$ avec $X \in N$ et $\gamma \in (N \cup \Gamma)^*$.

Une *dérivation immédiate* se note $\alpha \Rightarrow \beta$, avec $(\alpha, \beta) \in ((N \cup \Gamma)^*)^2$, et indique qu'il existe une règle de production $X \to \gamma$ de Π et des décompositions $\alpha = \alpha_1 X \alpha_2$ et $\beta = \alpha_1 \gamma \alpha_2$, avec $(\alpha_1, \alpha_2) \in ((N \cup \Gamma)^*)^2$. Une *dérivation* se note $\Rightarrow^*$ et indique l'existence d'une suite finie, éventuellement vide, de dérivations immédiates. Enfin, le *langage engendré par une grammaire* $\mathcal{G}$ se note $L(\mathcal{G})$ et désigne l'ensemble des mots de Γ^* qui dérivent du symbole initial S.

Selon les questions, nous représentons les mots de Γ par le type `char list` ou le type `string`.

Soit $\mathcal{G}_0$ la grammaire $(\{V\}, \{\mathtt{a}, \mathtt{b}, \mathtt{\#}\}, V, \Pi_0)$ dont les règles de production sont

$$\Pi_0 : \quad V \;\rightarrow\; \mathtt{a}\,V \mid \mathtt{b}\,V \mid \mathtt{\#}.$$

☐ 18 – Exhiber une expression régulière qui dénote le langage $L(\mathcal{G}_0)$.

☐ 19 – Dessiner l'automate de Glushkov associé à l'expression régulière de la question 18.

☐ 20 – Écrire une fonction OCaml `parseV (w : char list) : string * char list` dont la spécification suit :
Pré-condition : Il existe une décomposition du mot w en $w = vs$ avec $v \in L(\mathcal{G}_0)$ et $s \in \Sigma^*$.
Valeur de retour : Couple (v, s) où le préfixe v est représenté par le type `string` et s est le suffixe restant.
Effet : Une exception `SyntaxError` est levée si la pré-condition n'est pas satisfaite.

Soit $\mathcal{G}_1$ la grammaire $(\{T\}, \{\mathtt{(}, \mathtt{\_}, \mathtt{)}\}, T, \Pi_1)$ dont les règles de production sont

$$\Pi_1 : \quad T \;\rightarrow\; \mathtt{(}TT\mathtt{)} \mid \mathtt{\_}.$$

☐ 21 – Montrer que le langage $L(\mathcal{G}_1)$ est sans préfixe, c'est-à-dire qu'il n'existe pas deux mots non vides w et w' dans $L(\mathcal{G}_1)$ tels que w soit un préfixe strict de w'. On pourra, par exemple, raisonner par induction structurelle sur le nombre de parenthèses ouvrantes et fermantes qui se trouvent dans un préfixe d'un mot de $L(\mathcal{G}_1)$.

☐ 22 – Montrer que la grammaire $\mathcal{G}_1$ n'est pas ambiguë, c'est-à-dire que, pour tout mot du langage $L(\mathcal{G}_1)$, il n'existe qu'un seul arbre d'analyse (parfois aussi appelé arbre de dérivation) associé.

Soit $\mathcal{G}_2$ la grammaire $(\{T\}, \{\mathtt{var}, \mathtt{(}, \mathtt{)}, \mathtt{->}\}, T, \Pi_2)$ dont les règles de production sont

$$\Pi_2 : \quad T \;\rightarrow\; \mathtt{var} \mid \mathtt{(}TT\mathtt{)} \mid \mathtt{var\text{-}>}T.$$

☐ 23 – Montrer que la grammaire $\mathcal{G}_2$ n'est pas ambiguë.

Soit finalement $\mathcal{G}$ la grammaire $(\{T, V\}, \{\mathtt{a}, \mathtt{b}, \mathtt{\#}, \mathtt{(}, \mathtt{)}, \mathtt{->}\}, T, \Pi)$ dont les règles de production sont

$$\Pi : \quad \begin{cases} T & \rightarrow \quad V \mid \mathtt{(}TT\mathtt{)} \mid V\mathtt{\text{-}>}T \\ V & \rightarrow \quad \mathtt{a}\,V \mid \mathtt{b}\,V \mid \mathtt{\#}. \end{cases}$$

Nous admettons que la grammaire $\mathcal{G}$ n'est pas ambiguë. Nous appelons *variable* les mots de $L(\mathcal{G}_0)$. Informellement, la grammaire $\mathcal{G}$ engendre un langage $L(\mathcal{G})$ qui permet de parler de *variables*, d'*applications* d'une expression à une autre et de *fonctions*. Il nous reste à en construire une sémantique, ce qui sera fait en section 4.3.

Afin de représenter l'*arbre d'analyse* d'un mot du langage $L(\mathcal{G})$, nous introduisons le type `ada` par la déclaration suivante.

```
type ada = V of string | A of (ada * ada) | F of (string * ada)
```

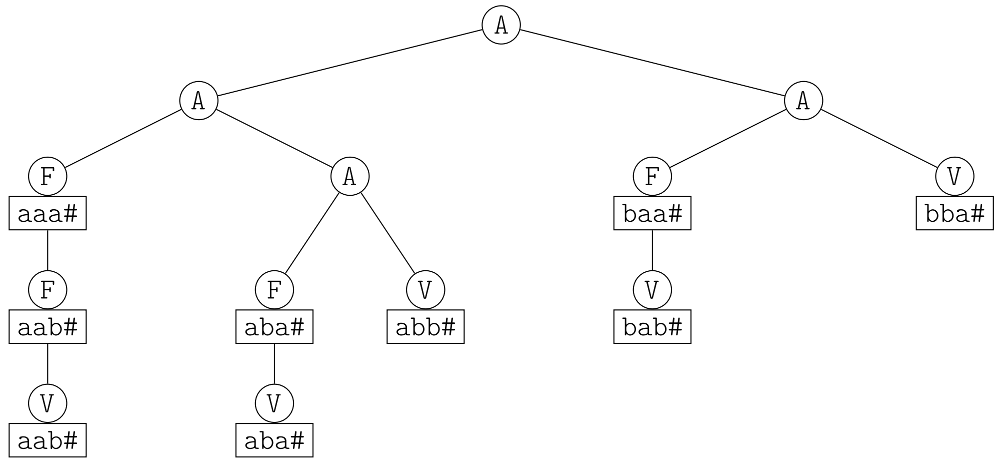
Nous notons $\mathscr{A}$ l'*ensemble des arbres d'analyse*.

Le constructeur OCaml `V` représente les dérivations immédiates de règle $T \to V$ et introduit les feuilles ; le constructeur OCaml `A` représente les dérivations immédiates de règle $T \to (TT)$ et introduit des nœuds internes d'arité 2 ; le constructeur OCaml `F` représente les dérivations immédiates de règle V`->`T et introduit des nœuds internes d'arité 1. Les dérivations à partir du symbole non terminal V sont directement représentées par une valeur OCaml de type `string`.

Par exemple, le mot

$$\texttt{((aaa\#->aab\#->aab\#(aba\#->aba\#abb\#))(baa\#->bab\#bba\#))} \in L(\mathcal{G})$$

admet pour arbre d'analyse :



☐ 24 – Écrire une fonction OCaml `parseT (w : char list) : ada * char list` dont la spécification suit :
Pré-condition : Il existe une décomposition du mot w en $w = ps$ avec $p \in L(\mathcal{G})$ et $s \in \Sigma^*$, dans laquelle le préfixe p est de longueur maximale.
Valeur de retour : Couple (a, s) où a est l'arbre d'analyse de p et s est le suffixe restant.
Effet : Une exception `SyntaxError` est levée si la pré-condition n'est pas satisfaite.

Indication OCaml : Nous supposons définie une fonction `charlist_of_string : string -> char list` qui transforme une chaîne de caractères en une liste de caractères.

☐ 25 – Écrire une fonction OCaml `parse (w : string) : ada` dont la valeur de retour est l'unique arbre d'analyse de w quand $w \in L(\mathcal{G})$ et qui lève une exception `SyntaxError` sinon.

Définition : Pour tout entier naturel $n \in \mathbb{N}$, nous définissons le mot

$$\lceil n \rceil = \texttt{b\#->a\#->(b\#(b\#...(b\#(b\#a\#))...))} \in L(\mathcal{G})$$

où la variable `b#` apparaît n fois à droite des deux symboles `->`.

☐ 26 – Indiquer s'il existe un automate fini capable de reconnaître le langage $\{\lceil n \rceil;\ n \in \mathbb{N}\}$ et justifier la réponse.

Définition : Plus généralement, nous disons qu'un mot w de $L(\mathcal{G})$ *incarne* un entier naturel $n \in \mathbb{N}$ s'il existe deux variables distinctes v_1 et v_2 dans $L(\mathcal{G}_0)$ telles que w est de la forme

$$w = v_2\texttt{->}v_1\texttt{->(}v_2\texttt{(}v_2\ldots\texttt{(}v_2\texttt{(}v_2v_1\texttt{))}\ldots\texttt{))} \in L(\mathcal{G})$$

où la variable v_2 apparaît n fois à droite des deux symboles `->`.

☐ 27 – Écrire une fonction OCaml `int_of_ada (a : ada) : int` dont la valeur de retour est l'entier naturel n incarné par a. Si un tel entier n'existe pas, une exception `SyntaxError` est levée.

4.2 Réécriture des variables et sérialisation

☐ 28 – Nommer une structure de donnée concrète efficace qui réalise le type de donnée abstrait « ensemble » lorsqu'il existe une relation d'ordre entre les objets à ranger.

Indication OCaml : Nous supposons définis un module OCaml `StringSet` permettant de construire des ensembles de chaînes de caractères persistants, de type `StringSet.t`, et des fonctions :

- `StringSet.mem : string -> StringSet.t -> bool` qui teste l'appartenance d'un élément à un ensemble.
- `StringSet.remove : string -> StringSet.t -> StringSet.t` qui retourne un ensemble privé d'un élément.
- `StringSet.singleton : string -> StringSet.t` qui construit un singleton à partir d'un élément.
- `StringSet.union : StringSet.t -> StringSet.t -> StringSet.t` qui construit l'union de deux ensembles.

Définition : L'ensemble des *variables libres* $VL(a) \subseteq \Sigma^*$ d'un arbre d'analyse $a \in \mathscr{A}$ est défini par induction structurelle avec les règles d'inférence suivantes :

- si l'arbre d'analyse a est de la forme `V v`, où $v \in L(\mathcal{G}_0)$, alors $VL(a)$ est le singleton $\{v\}$;
- si l'arbre d'analyse a est de la forme `A (a1,a2)`, où a_1 et a_2 sont deux arbres d'analyse, alors $VL(a)$ est la réunion $VL(a_1) \cup VL(a_2)$;
- si l'arbre d'analyse a est de la forme `F (v,a1)`, où $v \in L(\mathcal{G}_0)$ et a_1 est un arbre d'analyse, alors $VL(a)$ est l'ensemble $VL(a_1) \setminus \{v\}$.

☐ 29 – Écrire une fonction OCaml `free_vars (a : ada) : StringSet.t` dont la valeur de retour est l'ensemble des variables libres de l'arbre d'analyse a.

Définition : Un arbre d'analyse est dit *clos* s'il ne contient pas de variables libres ; de même, un mot de $L(\mathcal{G})$ est dit *clos* si son arbre d'analyse est clos.

Dans un arbre d'analyse clos $a \in \mathscr{A}$, pour toute chaîne de caractères $v \in L(\mathcal{G}_0)$, si la construction OCaml `V v` apparaît comme feuille de l'arbre a, alors il existe un nœud interne de la forme `F (v,_)` parmi les ascendants de `V v` qui coïncide avec l'« introduction » de la variable v.

Afin de ne plus s'encombrer avec des variables nommées par une chaîne de caractères, nous adoptons une nouvelle représentation des mots du langage $L(\mathcal{G})$ sous forme d'un arbre, appelé *terme de De Bruijn*.

Définition : Un terme de De Bruijn s'obtient à partir d'un arbre d'analyse $a \in \mathscr{A}$ en remplaçant toute feuille de l'arbre a, disons `V v` avec $v \in L(\mathcal{G}_0)$, par une feuille étiquetée par l'entier $u = 1 + \ell$, où $\ell \in \mathbb{N}$ est le nombre de nœuds internes de la forme `F (v',_)` avec $v' \in L(\mathcal{G}_0) \setminus \{v\}$, qui existent entre ladite feuille `V v` et le plus proche nœud interne de la forme `F (v,_)` parmi ses ascendants dans l'arbre d'analyse a.

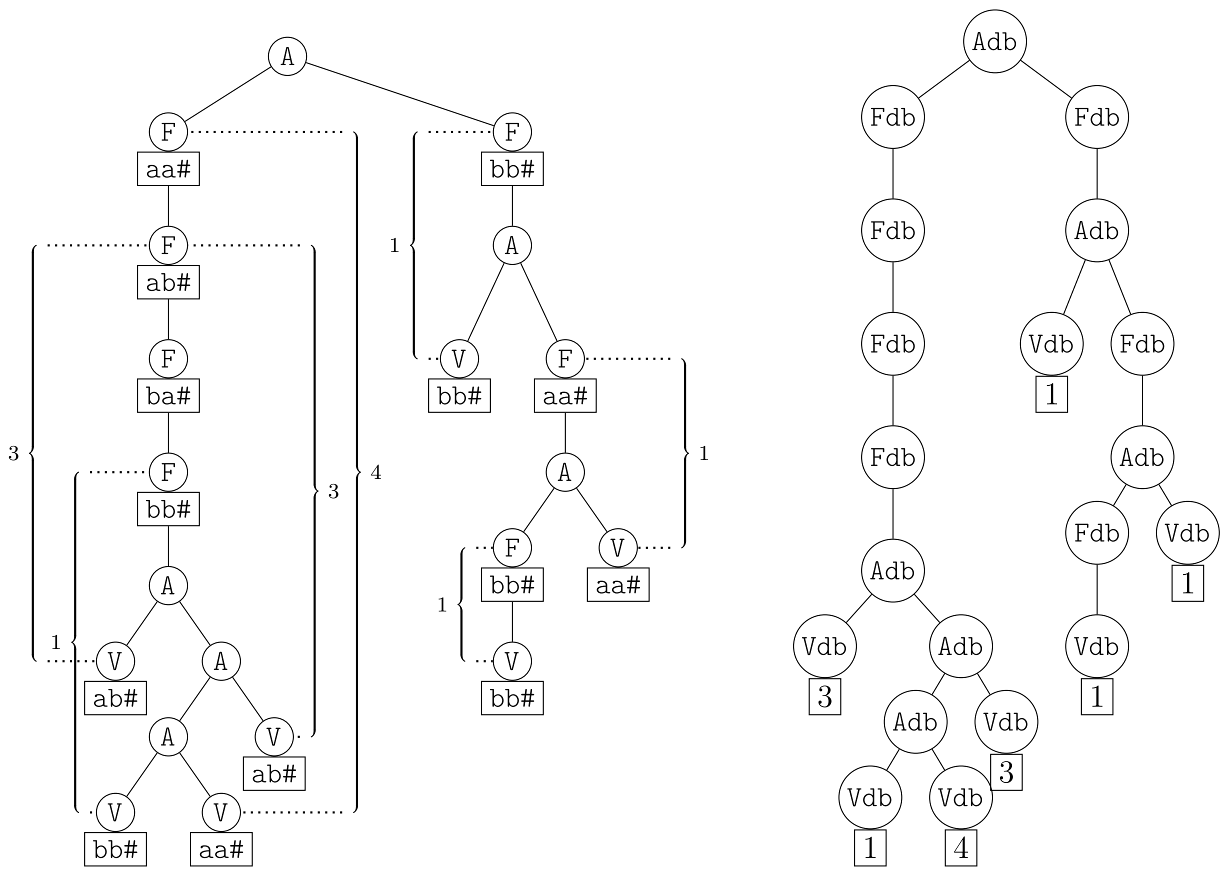
Nous déclarons un nouveau type

```
type tdb = Vdb of int | Adb of (tdb * tdb) | Fdb of tdb
```

Nous notons $\mathscr{T}$ l'*ensemble des termes de De Bruijn*.

Voici un exemple de construction d'un terme de De Bruijn (à droite) à partir d'un arbre d'analyse (à gauche) du mot

`(aa#->ab#->ba#->bb#->(ab#((bb#aa#)ab#))bb#->(bb#aa#->(bb#->bb#aa#)))` :



☐ 30 – Dessiner le terme de De Bruijn qui représente le mot $\lceil n \rceil$.

☐ 31 – Écrire une fonction OCaml `ada_of_tdb (t : tdb) : ada` dont la valeur de retour est un arbre d'analyse clos associé au terme de De Bruijn t.

Définition : Le *codage binaire* des termes de De Bruijn est l'application $t \in \mathscr{T} \mapsto \widehat{t} \in \{\mathtt{0}, \mathtt{1}\}^*$ définie par induction structurelle avec les règles d'inférence suivantes sur l'ensemble des termes de De Bruijn :

- si le terme $t \in \mathscr{T}$ est de la forme `Vdb u`, avec $u \in \mathbb{N}^\star$, alors $\widehat{t}$ est la chaîne de caractères `11..10` (avec le symbole `1` répété u fois).
- si le terme $t \in \mathscr{T}$ est de la forme `Adb (t1,t2)`, t_1 et t_2 étant deux termes de De Bruijn, alors $\widehat{t}$ est la chaîne de caractères $\mathtt{01}\widehat{t_1}\widehat{t_2}$.
- si le terme $t \in \mathscr{T}$ est de la forme `Fdb t1`, où t_1 est un terme de De Bruijn, alors $\widehat{t}$ est la chaîne de caractères $\mathtt{00}\widehat{t_1}$.

☐ 32 – Soient n un entier naturel et t le terme de De Bruijn associé au mot $\lceil n \rceil$ et obtenu à la question 30. Calculer la longueur $|\widehat{t}|$ de la chaîne de caractères qui encode t.

☐ 33 – Vérifier que le codage binaire des termes de De Bruijn est une application injective.

Nous utilisons le type `string` avec les caractères `'0'` et `'1'` afin de représenter des codages binaires de termes de De Bruijn.

☐ 34 – Écrire une fonction OCaml `decode (z : string) : tdb` dont la valeur de retour est l'unique terme de De Bruijn t tel que $\widehat{t} = z$ si un tel terme t existe et qui lève une exception `SyntaxError` sinon.
Il est possible de s'appuyer sur la fonction `charlist_of_string` précédemment présentée.

4.3 Interpréteur et décompresseur de De Bruijn

Dans cette sous-section, nous construisons un interpréteur, c'est-à-dire une procédure qui évalue les expressions appartenant au langage $L(\mathcal{G})$ et en déduisons une nouvelle description du mot $y_0 =$ `"1000000..."` relative à un décompresseur approprié.

Naïvement, lorsque nous rencontrons un sous-mot de la forme $w = (v\text{->}w_b\ w_a)$ avec $v \in L(\mathcal{G}_0)$ et $(w_a, w_b) \in (L(\mathcal{G}))^2$, nous aimerions que w soit équivalent à un mot tiré de w_b dont les occurrences de la variable v ont été remplacées par w_a. Autrement dit, lorsqu'un arbre d'analyse est de la forme `A (F (v, b), a)`, nous voudrions construire un nouvel arbre à partir de $b \in \mathscr{A}$ et dans lequel les apparitions de $v \in L(\mathcal{G}_0)$ sont devenues des sous-arbres $a \in \mathscr{A}$.

Définition : Soient $(a, b) \in (\mathscr{A})^2$ deux arbres d'analyse et $v \in L(\mathcal{G}_0)$ une variable. La *substitution de la variable v par l'arbre a dans l'arbre* b est l'application $[v \leftarrow a] : \mathscr{A} \to \mathscr{A}$ définie par induction structurelle avec les règles d'inférence suivantes :

— Si l'arbre $b \in \mathscr{A}$ est de la forme `V v1`, avec $v_1 \in L(\mathcal{G}_0)$, alors

$$[v \leftarrow a](b) = \begin{cases} \texttt{a} & \text{si } v = v_1 \\ \texttt{b} & \text{si } v \neq v_1. \end{cases}$$

— Si l'arbre $b \in \mathscr{A}$ est de la forme `A (b1, b2)`, où $b_1 \in \mathscr{A}$ et $b_2 \in \mathscr{A}$ sont deux arbres d'analyse, alors

$$[v \leftarrow a](b) = \texttt{A (b1',b2')}$$

où $b_1' = [v \leftarrow a](b_1)$ et $b_2' = [v \leftarrow a](b_2)$.

— Si l'arbre $b \in \mathscr{A}$ est de la forme `F (v1, b1)`, avec $v_1 \in L(\mathcal{G}_0)$ et $b_1 \in \mathscr{A}$, alors

$$[v \leftarrow a](b) = \begin{cases} \texttt{b} & & \text{si } v = v_1 \\ \texttt{F (v1,b1')} & \text{avec } b_1' = [v \leftarrow a](b_1) & \text{si } v \neq v_1 \text{ et } v_1 \notin VL(a) \\ \texttt{F (v1',b1')} & \text{avec } b_1' = [v \leftarrow a]([v_1 \leftarrow v_1'](b_1)) & \text{sinon} \end{cases}$$

où $v_1' \in L(\mathcal{G}_0)$ est une variable inédite qui n'appartient pas à $VL(a) \cup VL(b) \cup \{v, v_1\}$.

Nous supposons déjà programmée une fonction `new_string : unit -> string` inspirée de la section 3 qui permet, si besoin, d'engendrer une variable de $L(\mathcal{G}_0)$ jamais encore utilisée.

☐ 35 – Écrire une fonction OCaml `substitute (v : string) (by_a : ada) (in_b : ada) : ada` qui substitue la variable v par l'arbre a dans l'arbre b.

Définition : La *réduction en un pas* est l'application $\triangleright : \mathscr{A} \to \mathscr{A}$ définie par induction structuelle avec les règles d'inférence suivantes :

— Si l'arbre $a \in \mathscr{A}$ est de la forme `V v`, où $v \in L(\mathcal{G}_0)$, alors la valeur $\triangleright(a)$ n'est pas définie.

— Si l'arbre $a \in \mathscr{A}$ est de la forme `A (a1, a2)`, où $a_1 \in \mathscr{A}$ et $a_2 \in \mathscr{A}$ sont deux arbres d'analyse, alors

$$\triangleright(a) = \begin{cases} \texttt{A (a1, a2')} & \text{avec } a_2' = \triangleright(a_2) & \text{si } a_1 \text{ est de la forme } \texttt{V v} \\ \texttt{a'} & \text{avec } a' = [v \leftarrow a_2](a_{11}) & \text{si } a_1 \text{ est de la forme } \texttt{F (v, a11)} \\ \texttt{A (a1', a2)} & \text{avec } a_1' = \triangleright(a_1) & \text{sinon.} \end{cases}$$

— Si l'arbre $a \in \mathscr{A}$ est de la forme `F (v, a1)`, où $v \in L(\mathcal{G}_0)$ et $a_1 \in \mathscr{A}$, alors

$$\triangleright(a) = \texttt{F (v, a1')} \text{ où } a_1' = \triangleright(a_1).$$

☐ 36 – Écrire une fonction OCaml `reduce_one_step (a : ada) : ada` qui implémente la réduction en un pas $\triangleright$. Lorsque `reduce_one_step` rencontre un cas non défini, une exception `NoReduction` est levée.

☐ 37 – Écrire une fonction OCaml `interpret (a : ada) : ada` qui applique répétitivement la réduction en un pas $\triangleright$ à l'arbre d'analyse a jusqu'à ce que la réduction ne soit plus définie. La valeur de retour est le dernier arbre d'analyse rencontré.

Nous admettons que, pour tout entier naturel non nul n, si π est le mot de $L(\mathcal{G})$

$$\pi = \texttt{(a\#->((a\#a\#)a\#)}\lceil n \rceil\texttt{)},$$

l'expression `interpret (parse pi)` produit une incarnation de l'entier $n^{(n^n)}$. Nous notons $\mathcal{B}$ le décompresseur défini par

```
let decompB (z : string) : string =
  "string_of_int␣(int_of_ada␣(interpret␣(ada_of_tdb␣(decode␣" ^ z ^ "))))"
```

☐ 38 – Proposer une méthode pour obtenir une chaîne de caractères z_0, formée uniquement de `0` et de `1`, telle que `decompB z0` a pour valeur de retour la chaîne de caractères y_0 = `"1000000..."` et en calculer la longueur. En déduire que la complexité de Kolmogoroff de y_0 par rapport au décompresseur $\mathcal{B}$ vérifie

$$K_{\mathcal{B}}(y_0) \leq 70.$$

Fin de l'épreuve