Fondements de l'informatique

Les questions de programmation doivent être traitées en langage OCaml. On pourra utiliser toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`). L'utilisation d'autres modules est interdite.

Dans l'écriture d'une fonction, on pourra faire appel à des fonctions définies dans les questions précédentes, même si ces dernières n'ont pas été traitées. On pourra également définir des fonctions auxiliaires, mais il faudra alors préciser leurs rôles ainsi que le type et les significations de leurs arguments. Si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien satisfaites. Lorsque les choix d'implémentation ne découlent pas directement des spécifications de l'énoncé, il est conseillé de les expliquer.

On identifiera une même grandeur écrite dans deux polices de caractères différentes, en italique du point de vue mathématique (par exemple n) et en Computer Modern à chasse fixe du point de vue informatique (par exemple `n`).

Sans précision supplémentaire, lorsqu'une question demande la complexité d'une fonction, il s'agira de la complexité temporelle dans le pire des cas. La complexité sera exprimée sous la forme $\mathcal{O}(f(n))$ où n est la taille de l'argument de l'algorithme, et f une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement sauf mention spécifique dans la question.

Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

1 Compilation d'expressions vers une machine à pile

1.1 Une machine à pile

On s'intéresse à une machine à pile de type «calculatrice en notation polonaise inverse». Cette machine dispose de 4 instructions qui opèrent sur une pile contenant des valeurs entières :

$\mathsf{CONST}(n)$	empile l'entier n
$\mathsf{VAR}(X)$	empile la valeur de la variable X
ADD	dépile deux entiers n_1 et n_2, empile leur somme $n_1 + n_2$
MUL	dépile deux entiers n_1 et n_2, empile leur produit $n_1 \times n_2$

Un *code* pour la machine à pile est une liste d'instructions. L'exécution d'un code consiste à exécuter chaque instruction de la liste en séquence. Un *environnement*, donné en paramètre à la machine, associe une valeur à chaque variable. La machine s'arrête lorsque la dernière instruction a été exécutée. Le résultat de l'exécution est l'entier au sommet de la pile.

Exemple : l'exécution du code $\mathsf{CONST}(1); \mathsf{VAR}(X); \mathsf{ADD}$ dans un environnement où X vaut 2, et à partir d'une pile vide, effectue les étapes suivantes :

	Instruction exécutée	État de la pile après exécution de l'instruction
Étape 1	$\mathsf{CONST}(1)$	1
Étape 2	$\mathsf{VAR}(X)$	$2 \cdot 1$ (2 au sommet, 1 en dessous)
Étape 3	ADD	3

Le résultat de l'exécution est l'entier 3.

Pour simplifier, on supposera tout au long du sujet que l'exécution d'un code machine ne produit jamais d'erreurs : ni débordements arithmétiques, ni tentative de dépiler un entier depuis une pile vide.

Définition : On définit la *consommation en espace* d'un code comme la hauteur maximale atteinte par la pile pendant l'exécution du code, en partant de la pile vide.

Exemple : le code de l'exemple ci-dessus a une consommation en espace égale à 2 puisque juste avant l'exécution de l'instruction ADD, la pile contient deux valeurs, et en tout autre point la pile est vide ou contient une seule valeur.

Question 1 Dans un environnement où X vaut 2 et Y vaut 5, déterminer le résultat de l'exécution du code $\mathsf{VAR}(X); \mathsf{CONST}(1); \mathsf{ADD}; \mathsf{VAR}(Y); \mathsf{MUL}$. Préciser la consommation en espace de ce code.

Une instruction est représentée en OCaml par le type `instr` suivant :

```
type instr = CONST of int | VAR of string | ADD | MUL
```

Un code est alors représenté par une liste d'éléments de type `instr`. L'environnement est représenté par un objet de type `string -> int`, c'est-à-dire une fonction qui prend en argument une chaîne de caractère correspondant à une variable et renvoie un entier.

Question 2 Écrire une fonction `exec : instr list -> (string -> int) -> int` qui prend en argument un code et un environnement et renvoie le résultat de l'exécution du code à partir d'une pile vide.

Question 3 Écrire une fonction `conso_espace : instr list -> int` qui calcule la consommation en espace d'un code donné en argument.

1.2 Compilation simple des expressions arithmétiques

On s'intéresse aux expressions arithmétiques formées à partir de constantes entières $0, 1, 2, \ldots, n, \ldots$ et de variables symboliques $X, Y, Z, \ldots$ en utilisant les opérateurs $+$ (somme) et $\times$ (produit).

Ces expressions sont représentées par leurs arbres de syntaxe abstraite : des arbres binaires avec, à chaque feuille, une constante ou une variable et, à chaque nœud, un opérateur $+$ ou $\times$. Par exemple, l'arbre A représenté en figure 1 correspond au produit de X plus 1 et de Y :

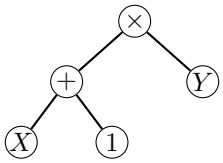


FIGURE 1 – L'expression A

La sémantique de ce langage d'expressions arithmétiques est donnée par les équations suivantes, qui définissent la valeur entière $[\![e]\!]_\rho$ d'une expression e dans un environnement ρ.

$$[\![\textcircled{n}]\!]_\rho = n \qquad [\![\textcircled{X}]\!]_\rho = \rho(X)$$

$$[\![\overset{+}{\underset{e_1}{\diagdown}\,\underset{e_2}{\diagup}}]\!]_\rho = [\![e_1]\!]_\rho + [\![e_2]\!]_\rho \qquad [\![\overset{\times}{\underset{e_1}{\diagdown}\,\underset{e_2}{\diagup}}]\!]_\rho = [\![e_1]\!]_\rho \times [\![e_2]\!]_\rho$$

On rappelle l'algorithme de compilation classique $\mathcal{C}$ qui, à une expression e représentée par son arbre, associe un code $\mathcal{C}(e)$ de la machine à pile :

$$\mathcal{C}(\textcircled{n}) = \mathsf{CONST}(n) \qquad \mathcal{C}(\textcircled{X}) = \mathsf{VAR}(X)$$

$$\mathcal{C}(\overset{+}{\underset{e_1}{\diagdown}\,\underset{e_2}{\diagup}}) = \mathcal{C}(e_1); \mathcal{C}(e_2); \mathsf{ADD} \qquad \mathcal{C}(\overset{\times}{\underset{e_1}{\diagdown}\,\underset{e_2}{\diagup}}) = \mathcal{C}(e_1); \mathcal{C}(e_2); \mathsf{MUL}$$

Question 4 Quel code est produit par $\mathcal{C}$ pour l'expression A de la figure 1 ?

Question 5 Soit e une expression, P la pile de la machine, et ρ un environnement. On fait exécuter par la machine le code compilé $\mathcal{C}(e)$ avec ρ comme environnement et P comme pile initiale. Démontrer par récurrence sur la structure de e que la pile finale, lorsque la machine s'arrête, est $[\![e]\!]_\rho \cdot P$, c'est à dire la pile contenant la valeur $[\![e]\!]_\rho$ au sommet et les mêmes valeurs que P en dessous.

Question 6 Dans cette question et la question 16 uniquement, on ajoute à notre langage d'expressions la soustraction $e_1 - e_2$, représentée par un arbre de la forme $\overset{\ominus}{\underset{e_1}{\diagup}\underset{e_2}{\diagdown}}$, ainsi que le calcul de l'opposé $-e$, représenté par un arbre de la forme $\overset{\ominus}{\underset{e}{|}}$. En parallèle, on ajoute à notre machine à pile une instruction OPP qui change le signe de l'entier au sommet de la pile. On veut étendre l'algorithme de compilation $\mathcal{C}$ pour traiter la soustraction et l'opposé. Donner les équations qui définissent $\mathcal{C}(\overset{\ominus}{\underset{e}{|}})$ et $\mathcal{C}(\overset{\ominus}{\underset{e_1}{\diagup}\underset{e_2}{\diagdown}})$, et les justifier informellement.

1.3 Consommation en espace du code compilé

Question 7 On considère les deux familles d'expressions G_n et D_n pour $n \geqslant 0$ définies par les récurrences suivantes :

$$G_0 = \textcircled{0} \qquad\qquad D_0 = \textcircled{0}$$

$$G_n = \overset{\oplus}{\underset{G_{n-1}}{\diagup}\underset{n}{\diagdown}} \text{ si } n > 0 \qquad\qquad D_n = \overset{\oplus}{\underset{n}{\diagup}\underset{D_{n-1}}{\diagdown}} \text{ si } n > 0$$

Montrer que pour tout $n \geqslant 1$, le code compilé $\mathcal{C}(G_n)$ a une consommation en espace égale à 2, et que pour tout $n \geqslant 1$ le code compilé $\mathcal{C}(D_n)$ a une consommation en espace égale à $n + 1$.

Question 8 Pour une expression e quelconque, on note $\mathcal{T}(e)$ la consommation en espace du code compilé $\mathcal{C}(e)$. Donner et justifier les relations de récurrence permettant de calculer $\mathcal{T}(\textcircled{n})$, $\mathcal{T}(\textcircled{X})$, $\mathcal{T}(\overset{\oplus}{\underset{e_1}{\diagup}\underset{e_2}{\diagdown}})$ et $\mathcal{T}(\overset{\otimes}{\underset{e_1}{\diagup}\underset{e_2}{\diagdown}})$ en fonction de $\mathcal{T}(e_1)$ et $\mathcal{T}(e_2)$.

1.4 Compilation qui minimise la consommation en espace

L'informaticien russe Andreï Erchov propose un algorithme de compilation en 1957 : l'idée centrale est, dans le cas d'une somme ou d'un produit de deux expressions e_1 et e_2, de traiter en premier la sous-expression e_i qui a le plus grand nombre de Strahler $\mathcal{S}(e_i)$, le nombre de Strahler d'une expression étant défini par la récurrence suivante :

$$\mathcal{S}(\textcircled{n}) = \mathcal{S}(\textcircled{X}) = 1$$

$$\mathcal{S}(\overset{\oplus}{\underset{e_1}{\diagup}\underset{e_2}{\diagdown}}) = \mathcal{S}(\overset{\otimes}{\underset{e_1}{\diagup}\underset{e_2}{\diagdown}}) = \begin{cases} 1 + \mathcal{S}(e_1) & \text{si } \mathcal{S}(e_1) = \mathcal{S}(e_2) \\ \max(\mathcal{S}(e_1), \mathcal{S}(e_2)) & \text{si } \mathcal{S}(e_1) \neq \mathcal{S}(e_2) \end{cases}$$

Ces nombres ont été introduits par le géographe américain Arthur N. Strahler en 1952 pour l'étude des réseaux hydrographiques des cours d'eau.

L'algorithme de compilation d'Erchov $\mathcal{E}$ considéré est alors :

$$\mathcal{E}(\textcircled{n}) = \mathsf{CONST}(n) \qquad\qquad \mathcal{E}(\textcircled{X}) = \mathsf{VAR}(X)$$

$$\mathcal{E}(\overset{\oplus}{\underset{e_1}{\diagup}\underset{e_2}{\diagdown}}) = \begin{cases} \mathcal{E}(e_1); \mathcal{E}(e_2); \mathsf{ADD} & \text{si } \mathcal{S}(e_1) \geqslant \mathcal{S}(e_2) \\ \mathcal{E}(e_2); \mathcal{E}(e_1); \mathsf{ADD} & \text{si } \mathcal{S}(e_1) < \mathcal{S}(e_2) \end{cases} \quad \mathcal{E}(\overset{\otimes}{\underset{e_1}{\diagup}\underset{e_2}{\diagdown}}) = \begin{cases} \mathcal{E}(e_1); \mathcal{E}(e_2); \mathsf{MUL} & \text{si } \mathcal{S}(e_1) \geqslant \mathcal{S}(e_2) \\ \mathcal{E}(e_2); \mathcal{E}(e_1); \mathsf{MUL} & \text{si } \mathcal{S}(e_1) < \mathcal{S}(e_2) \end{cases}$$

Question 9 Montrer l'analogue de la question 5 pour l'algorithme de compilation d'Erchov : si on fait exécuter par la machine le code compilé $\mathcal{E}(e)$ avec ρ comme environnement et P comme pile initiale, la machine s'arrête avec la pile finale $[\![e]\!]_\rho \cdot P$. On ne demande pas de refaire une démonstration complète, mais juste d'expliquer ce qui change par rapport à la démonstration de la question 5.

Question 10 Calculer les nombres de Strahler $\mathcal{S}(G_n)$ et $\mathcal{S}(D_n)$ pour les expressions G_n et D_n définies à la question 7.

Question 11 En déduire la forme des codes $\mathcal{E}(G_n)$ et $\mathcal{E}(D_n)$ produits par l'algorithme d'Erchov. Quelle est la consommation en espace de ces codes ?

Question 12 Démontrer, pour toute expression e, que la consommation en espace du code compilé $\mathcal{E}(e)$ est égale à $\mathcal{S}(e)$. En d'autres termes, le nombre de Strahler est la consommation en espace du code produit par l'algorithme d'Erchov.

Question 13 On définit la taille $\|e\|$ d'une expression e comme le nombre de nœuds et de feuilles dans son arbre de syntaxe abstraite. Démontrer l'inégalité suivante :

$$\mathcal{S}(e) \leqslant \log_2(\|e\| + 1)$$

Question 14 En déduire qu'une machine disposant de 4 emplacements de pile peut évaluer n'importe quelle expression contenant au plus 7 additions ou multiplications.

Question 15 Expliquer comment construire une expression e de taille minimale dont le code compilé $\mathcal{E}(e)$ ne peut pas s'exécuter sur une machine disposant de 4 emplacements de pile. Quelle est la taille de cette expression ? Pourquoi cette taille est-elle minimale parmi toutes les expressions e satisfaisant cette propriété ?

Question 16 Comme dans la question 6, on ajoute à notre langage d'expressions l'opposé $\ominus e$ et la différence $e_1 \ominus e_2$. Toujours comme dans la question 6, on ajoute aussi l'instruction OPP à notre machine à pile. Étendre la définition des nombres de Strahler et l'algorithme d'Erchov aux deux nouvelles formes d'expressions. On donnera les équations qui définissent les nombres $\mathcal{S}(e_1 \ominus e_2)$ et $\mathcal{S}(\ominus e)$, ainsi que les codes $\mathcal{E}(e_1 \ominus e_2)$ et $\mathcal{E}(\ominus e)$.

On représente une expression en OCaml par le type `expr` suivant :

```
type expr =
  | Const of int
  | Var of string
  | Somme of expr * expr
  | Produit of expr * expr
```

Question 17 Écrire une fonction `strahler : expr -> int` qui calcule le nombre de Strahler d'une expression donnée en argument. Cette fonction devra avoir une complexité linéaire en la taille de son argument, et on demande de prouver cette complexité.

Question 18 Écrire une fonction `erchov : expr -> instr list` qui prend en argument une instruction et calcule le code associé selon l'algorithme d'Erchov.

Question 19 Déterminer la complexité temporelle de la fonction `erchov`.

Question 20 Comment modifier la fonction `erchov` pour que sa complexité temporelle soit linéaire en la taille de son argument (si ce n'est pas déjà le cas) ? On ne demande pas de réécrire le code mais de décrire les modifications en français.

2 Compilation d'expressions vers une machine à registres

2.1 Une machine à registres

À la place de la machine à pile de la partie 1, nous allons maintenant cibler une machine à registres. Cette machine dispose de K registres nommés $R_1, \ldots, R_K$, chaque registre pouvant contenir une valeur entière. La machine dispose de 4 instructions :

$\mathsf{MOV}(R_i, n)$	met l'entier n dans le registre R_i
$\mathsf{LOAD}(R_i, X)$	charge la valeur de la variable X depuis la mémoire et la met dans le registre R_i
$\mathsf{ADD}(R_i, R_j, R_k)$	met dans R_i la somme des valeurs des registres R_j et R_k
$\mathsf{MUL}(R_i, R_j, R_k)$	met dans R_i le produit des valeurs des registres R_j et R_k

Un code pour la machine à registres est une liste d'instructions, qui sont exécutées en séquence.

Exemple : l'exécution du code $\mathsf{LOAD}(R_1, X); \mathsf{MOV}(R_2, 1); \mathsf{ADD}(R_1, R_1, R_2)$ a pour effet de mettre la valeur de l'expression $X + 1$ dans le registre R_1, et l'entier 1 dans le registre R_2. Les autres registres ne sont pas modifiés.

2.2 Compilation des expressions arithmétiques

On peut adapter les algorithmes de la partie 1 en traitant les K registres de la machine comme une pile de hauteur au plus K. Pour représenter une pile de hauteur $k \leqslant K$, on stocke le sommet de la pile dans le registre R_k, l'élément suivant de la pile dans le registre R_{k-1}, jusqu'à la base de la pile qui est stockée dans R_1.

En appliquant cette technique à l'algorithme de compilation simple de la section 1.2, on obtient l'algorithme de compilation $\mathcal{C}$ suivant, qui prend en arguments une expression e et un numéro de registre d :

$$\mathcal{C}(\textcircled{n}, d) = \mathsf{MOV}(R_d, n)$$
$$\mathcal{C}(\textcircled{X}, d) = \mathsf{LOAD}(R_d, X)$$
$$\mathcal{C}(\ \overset{e_1}{\diagdown}\!\textcircled{+}\!\overset{e_2}{\diagup}\ , d) = \mathcal{C}(e_1, d); \mathcal{C}(e_2, d+1); \mathsf{ADD}(R_d, R_d, R_{d+1})$$
$$\mathcal{C}(\ \overset{e_1}{\diagdown}\!\textcircled{\times}\!\overset{e_2}{\diagup}\ , d) = \mathcal{C}(e_1, d); \mathcal{C}(e_2, d+1); \mathsf{MUL}(R_d, R_d, R_{d+1})$$

Question 21 Quel code est produit par $\mathcal{C}(A, 1)$ pour l'expression A représentée figure 1 ?

Question 22 Pour e une expression et d un entier, expliquer informellement l'effet du code produit par $\mathcal{C}(e, d)$ sur les registres de la machine : une fois exécuté ce code, quelle valeur contient le registre R_d ? et le registre R_i pour $i < d$?

Question 23 Donner un exemple d'expression e telle que le code compilé $\mathcal{C}(e, 1)$ est incorrect car utilisant trop de registres : le code utilise le registre R_{K+1} qui n'existe pas.

Question 24 En s'inspirant de l'algorithme d'Erchov, définir un algorithme de compilation $\mathcal{E}$ prenant en entrée e et d et produisant du code de la machine à registre qui :

1. calcule la valeur de l'expression e et met cette valeur dans le registre R_d ;
2. utilise au plus $\mathcal{S}(e)$ registres différents.

En déduire qu'on peut compiler correctement toutes les expressions dont le nombre de Strahler est au plus K.

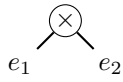
2.3 Optimalité en nombre de registres utilisés

Dans cette section, nous allons montrer la réciproque de la question précédente : tout code de la machine à registres qui évalue correctement une expression e a besoin d'au moins $\mathcal{S}(e)$ registres différents. Par conséquent, l'algorithme de la question précédente est optimal en nombre de registres utilisés. Ce résultat a été publié par Ravi Sethi et Jeffrey D. Ullman en 1970.

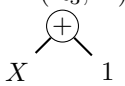
Définition : Soient c un code de la machine à registres (une liste d'instructions), e une expression, et R un registre. On dit que c *calcule la valeur de* e *dans* R si les conditions suivantes sont remplies :

— c est de la forme $c_1; I; c_2$ où c_1 et c_2 sont des listes d'instructions et I une seule instruction ;
— aucune instruction de la liste c_2 n'écrit dans le registre R ;
— si e est une constante n, l'instruction I est $\mathsf{MOV}(R, n)$;
— si e est une variable X, l'instruction I est $\mathsf{LOAD}(R, X)$;
— si e est une somme $\overset{e_1}{\diagdown}\!\textcircled{+}\!\overset{e_2}{\diagup}$, l'instruction I est de la forme $\mathsf{ADD}(R, R', R'')$, et de plus le code c_1 calcule la valeur de e_1 dans R' et la valeur de e_2 dans R'' ;

Tournez la page S.V.P.

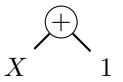
— si e est un produit 

, l'instruction I est de la forme $\mathsf{MUL}(R, R', R'')$, et de plus le code c_1 calcule la valeur de e_1 dans R' et la valeur de e_2 dans R''.

Exemple : le code $c = \mathsf{MOV}(R_2, 1); \mathsf{LOAD}(R_3, X); \mathsf{ADD}(R_2, R_3, R_2)$ calcule la valeur de la variable X dans le registre R_3 et la valeur de la somme 

dans le registre R_2. Mais il ne calcule pas la valeur de la constante 1 dans R_2.

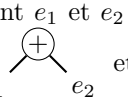
Définition : On dit qu'un code c qui calcule la valeur d'une expression e *utilise au moins k registres* s'il existe un point dans le code c où k registres différents contiennent des résultats intermédiaires du calcul de e, c'est-à-dire les valeurs de k sous-expressions de e.

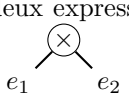
Exemple : le code c de l'exemple précédent utilise au moins deux registres, car au point qui suit immédiatement l'instruction LOAD, les registres R_2 et R_3 contiennent des résultats intermédiaires du calcul de 

, à savoir les valeurs de 1 et de X.

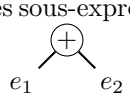
Définition : On dit qu'*une expression e nécessite au moins k registres* si tout code machine c qui calcule la valeur de e (dans un registre quelconque) utilise au moins k registres.

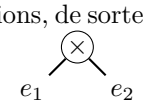
Question 25 En utilisant les résultats de la question 22, montrer que le code $\mathcal{C}(e, d)$ produit par l'algorithme simple de la section 2.2 calcule la valeur de e dans R_d (au sens de la définition ci-dessus).

Question 26 Soient e_1 et e_2 deux expressions telles que e_1 **ou** e_2 nécessite au moins k registres. Montrer que les expressions 

et 

nécessitent au moins k registres.

Question 27 Soient e_1 et e_2 deux expressions telles que e_1 **et** e_2 nécessitent au moins k registres. On suppose que les valeurs de e_1 et de ses sous-expressions sont différentes des valeurs de e_2 et de ses sous-expressions, de sorte qu'aucun résultat intermédiaire ne peut être réutilisé. Montrer que les expressions 

et 

nécessitent au moins $k + 1$ registres.

Question 28 Déduire des deux questions précédentes que toute expression e sans partage de sous-expressions nécessite au moins $\mathcal{S}(e)$ registres.

3 Compilation d'expressions avec partage

3.1 Introduction du problème

Lors de la compilation d'une expression, une même sous-expression peut apparaître plusieurs fois. Par exemple, dans l'expression e_0 suivante :

$$(((3 \times 5) + 2) + (3 \times 5)) \times ((3 \times 5) + 2)$$

la sous-expression 3×5 apparaît trois fois, et la sous-expression $(3 \times 5) + 2$ apparaît deux fois. En s'autorisant à ne calculer qu'une seule fois certaines de ces sous-expressions, on peut améliorer l'efficacité du calcul, en temps (nombre d'évaluations de sous-expressions) et en espace (nombre maximal de registres utilisés simultanément). On représente donc dorénavant une expression non plus par un arbre, mais par un graphe orienté sans cycle.

Question 29 En utilisant le graphe H_0, montrer qu'il existe une suite d'instructions sur une machine à registres qui compile e_0 en utilisant strictement moins de registres et strictement moins d'instructions que la compilation de l'arbre $\mathcal{A}_0$.

Définition : On définit un *graphe orienté* $G = (S, A)$ comme un couple composé d'un ensemble fini non vide de *sommets* S et un ensemble d'*arêtes* $A \subset S \times S$.

Pour $(s, t) \in A$, on dit que t est un *successeur* de s et que s est un *prédécesseur* de t. Pour $(s, t) \in S^2$, un *chemin* de s à t est une suite finie $c = (s_0, s_1, \ldots, s_k)$ d'éléments de S, où $s_0 = s$, $s_k = t$, et pour $i \in \{0, 1, \ldots, k-1\}$, s_{i+1} est un successeur de s_i. On dit alors que k est la *longueur* du chemin c. Si de plus $k \geqslant 1$ et $s = t$, on dit que c est un *cycle*.

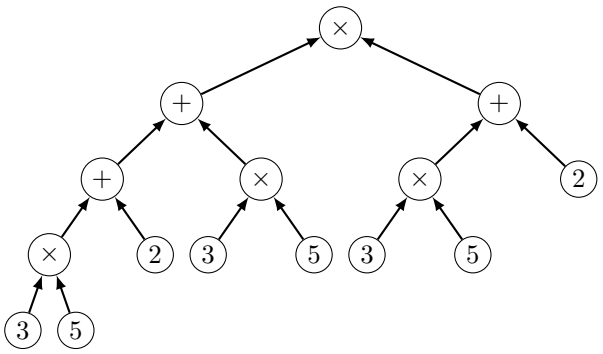


FIGURE 2 – e_0 sous forme d'un arbre $\mathcal{A}_0$

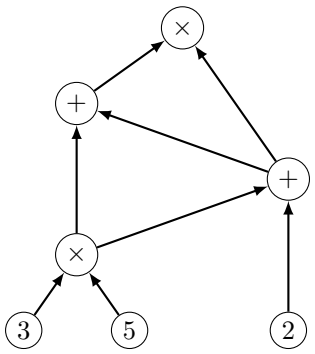


FIGURE 3 – e_0 sous forme d'un graphe H_0

On appelle *degré sortant* d'un sommet $s \in S$ son nombre de successeurs et *degré entrant* de s son nombre de prédécesseurs. On dit que s est un *puits* s'il est de degré sortant nul, et une *source* s'il est de degré entrant nul.

Pour la suite de ce problème, on ne considérera que des graphes orientés sans cycle.

On cherche à établir une heuristique pour déterminer l'ordre dans lequel doivent être calculés les sommets d'un graphe correspondant à une expression. Dans la mesure où cet ordre ne dépend pas des opérations, on représente un graphe en OCaml sans se soucier de l'étiquette des sommets.

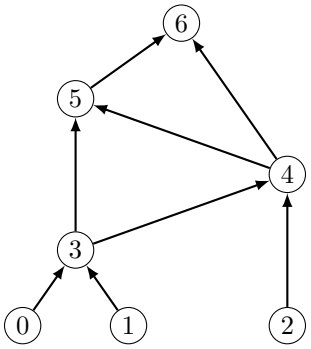


FIGURE 4 – Représentation de H_0 sans étiquette, en numérotant les sommets

Le type OCaml utilisé est le suivant :

```
type graphe = int list array
```

Dans le détail, pour $n \geqslant 0$, un graphe $G = (S, A)$ à n sommets sera représenté par un tableau `g` de taille n, où on assimilera S à $\{0, 1, \ldots, n-1\}$, et tel que pour $s \in S$, `g.(s)` est une liste qui contient tous les successeurs du sommet s (dans un ordre arbitraire et sans doublon). Par exemple, le graphe H_0 précédent peut être créé par la commande :

```
let h0 = [| [3]; [3]; [4]; [5; 4]; [5; 6]; [6]; [] |]
```

On remarque que lors de la compilation, il est intéressant de savoir quels sont les successeurs d'un sommet s donné (quels sont les sommets dont le calcul dépend de s ? À quel moment le registre qui stocke la valeur de s peut-il être libéré ?), ainsi que les prédécesseurs d'un sommet s (quels sont les sommets dont le calcul est nécessaire pour calculer s ?). Pour un graphe $G = (S, A)$, on appelle *graphe transposé* de G, noté ${}^tG$, le graphe où on a « retourné » toutes les arêtes, c'est-à-dire le graphe ${}^tG = (S, A')$ où $A' = \{(s,t), (t,s) \in A\}$.

Question 30 Écrire une fonction `transpose : graphe -> graphe` qui prend en argument un graphe $G = (S, A)$ et renvoie le graphe ${}^tG$. La complexité de cette fonction devra être en $\mathcal{O}(|S| + |A|)$ et on demande de prouver cette complexité.

Définition : Pour un graphe $G = (S, A)$ à n sommets, un *ordre topologique* $(s_0, s_1, \ldots, s_{n-1})$ est une suite de sommets de taille n et sans doublon telle que pour $i, j \in \{0, 1, \ldots, n-1\}^2$, $(s_i, s_j) \in A \Rightarrow i < j$. Autrement dit, un sommet apparaît toujours dans l'ordre avant ses successeurs.

Pour un graphe G à n sommets, on représente en OCaml un ordre topologique $(s_0, s_1, \ldots, s_{n-1})$ par un tableau de taille n contenant les valeurs s_0, s_1, …, s_{n-1}. Par exemple, le tableau suivant représente un ordre topologique de H_0 :

```
let topo = [|2; 1; 0; 3; 4; 5; 6|]
```

Question 31 On considère la fonction `est_ordre : graphe -> int array -> bool` suivante, qui prend en argument un graphe `g` correspondant à un graphe à n sommets et un tableau `topo` de taille n contenant tous les entiers de $\{0, 1, \ldots, n-1\}$ sans doublon.

```
let est_ordre g topo =
    let n = Array.length topo in
    let rang = Array.make n (-1) in
    for i = 0 to n - 1 do
        rang.(topo.(i)) <- i
    done;
    let rec verif_aretes s lst =
        match lst with
        | []     -> s = n - 1 || verif_aretes (s + 1) g.(s + 1)
        | t :: q -> rang.(s) < rang.(t) && verif_aretes s q in
    verif_aretes 0 g.(0)
```

On cherche à montrer que cette fonction renvoie `true` si `topo` est un ordre topologique valide pour `g` et `false` sinon. Expliquer et caractériser par une assertion logique ce que contient le tableau `rang` à la fin de la boucle `for`. Donner une précondition nécessaire et suffisante pour que l'appel `verif_aretes s lst` renvoie `true`. Conclure que la fonction `est_ordre` a le comportement attendu.

Pour un ordre topologique donné, on compile un graphe de la manière suivante :
- les sommets du graphe sont calculés dans l'ordre topologique ;
- lorsqu'un sommet s_j est calculé :
  - pour chaque prédécesseur s_i de s_j ($i < j$), si s_j est le dernier successeur de s_i apparaissant dans l'ordre topologique, le registre stockant s_i peut être libéré au moment du calcul ;
  - on utilise alors un registre (libéré si possible, ou nouveau sinon) pour stocker le résultat de s_j ;
  - si s_j est un puits, on libère ensuite ce registre.

En particulier, chaque sommet ne sera calculé qu'une seule fois.

Question 32 Donner le nombre de registres utilisés pour calculer H_0 selon l'ordre $(2, 1, 0, 3, 4, 5, 6)$. Donner un ordre topologique utilisant strictement moins de registres.

On cherche à déterminer une heuristique permettant de calculer un ordre topologique utilisant peu de registres.

Définition : On redéfinit le *nombre de Strahler* d'un sommet de la manière suivante :
- si s est une source, alors $\mathcal{S}(s) = 1$;
- si s possède un unique prédécesseur t, alors $\mathcal{S}(s) = \mathcal{S}(t)$;
- si s possède deux prédécesseurs t et u, alors $\mathcal{S}(s) = \begin{cases} 1 + \mathcal{S}(t) & \text{si } \mathcal{S}(t) = \mathcal{S}(u) \\ \max(\mathcal{S}(t), \mathcal{S}(u)) & \text{si } \mathcal{S}(t) \neq \mathcal{S}(u) \end{cases}$

Pour les deux questions suivantes, on supposera que les graphes donnés en arguments correspondent à des expressions écrites avec des opérateurs binaires, c'est-à-dire possédant un unique puits et dont tous les sommets ont un degré entrant inférieur ou égal à 2.

Question 33 Écrire une fonction `strahler_graphe : graphe -> int array` qui prend en argument un graphe G et renvoie un tableau `st` tel que pour tout sommet s, `st.(s)` est égal à $\mathcal{S}(s)$. La complexité de cette fonction devra être en $\mathcal{O}(|S| + |A|)$ mais on ne demande pas de le prouver.

L'heuristique proposée pour trouver un ordre topologique particulier, appelé *ordre de Strahler*, pour un sommet s d'un graphe G est la suivante :

— si s est une source, l'ordre de Strahler est (s) ;
— si s possède un unique prédécesseur t, alors l'ordre de Strahler de s est l'ordre de Strahler de t suivi de s ;
— si s possède deux prédécesseurs t et u tels que $\mathcal{S}(t) \geqslant \mathcal{S}(u)$, alors l'ordre de Strahler de s est l'ordre de Strahler de t suivi de l'ordre de Strahler de u, auquel on a enlevé les sommets apparaissant dans l'ordre de Strahler de t, suivi de s.

L'ordre de Strahler d'un graphe G est alors l'ordre de Strahler de son unique puits.

Question 34 Écrire une fonction `ordre_strahler : graphe -> int array` qui prend en argument un graphe G et renvoie l'ordre de Strahler de G.

Question 35 En supposant qu'on commence toujours par le sommet de plus petit indice lorsqu'on a le choix entre deux sommets, donner l'ordre de Strahler déterminé par l'heuristique précédente sur le graphe de la figure 5 et le nombre de registres utilisés par cet ordre. Montrer qu'il existe un ordre topologique utilisant strictement moins de registres que l'ordre précédent.

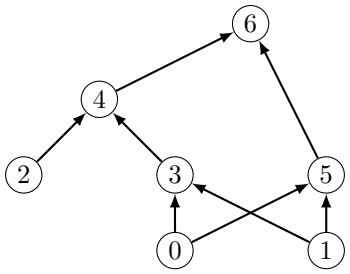


FIGURE 5

3.2 Jeu du marquage

Pour un graphe $G = (S, A)$ de $\mathscr{G}$, on définit le *jeu du marquage* sur G comme un jeu à un joueur. Les mouvements autorisés dans ce jeu sont les suivants :

(i) enlever un marqueur d'un sommet ;
(ii) si tous les prédécesseurs d'un sommet s ont un marqueur, marquer s ;
(iii) si tous les prédécesseurs d'un sommet s ont un marqueur, déplacer un marqueur de l'un de ces prédécesseurs vers s.

Le jeu commence sans aucun sommet marqué, et l'objectif du jeu est que chaque puits G soit marqué au moins une fois (pas nécessairement simultanément). On remarque qu'un cas particulier du mouvement (ii) est qu'une source peut être marquée à tout moment.

L'intérêt de ce jeu est qu'il représente la compilation d'un graphe orienté sans cycle : la libération d'un registre correspond au mouvement (i), et le calcul d'une expression en fonction de ses sous-expressions directes au mouvement (ii) si un nouveau registre est utilisé pour stocker le résultat, ou au mouvement (iii) si on utilise le registre de l'une des sous-expressions.

Définition : Une *stratégie* pour un marquage de G est une suite de mouvements permettant de gagner le jeu. Le *temps* de cette stratégie est le nombre de mouvements (ii) et (iii) utilisés, et l'*espace* de cette stratégie est le nombre maximal de marqueurs utilisés simultanément. On notera $M_{\min}(G)$ l'espace minimal d'une stratégie d'un marquage de G, c'est-à-dire le nombre minimal de marqueurs nécessaires pour marquer tous les puits de G.

Question 36 Déterminer $M_{\min}(H_0)$, H_0 étant le graphe représenté en figure 4.

Question 37 Montrer que tout graphe à n sommets possède une stratégie en temps n et en espace n.

Définition : Pour $k > 0$, on définit le *graphe pyramide* P_k par :

— P_1 est le graphe à un seul sommet, sans arête ;
— pour $k > 1$, si P_k est défini, et que $\{s_1, s_2, \ldots, s_k\}$ sont les sources de P_k, alors on définit P_{k+1} en rajoutant $k+1$ sommets $\{t_0, \ldots, t_k\}$ et les $2k$ arêtes (t_{i-1}, s_i) et (t_i, s_i), pour $i \in \{1, 2, \ldots, k\}$.

Tournez la page S.V.P.

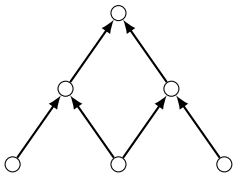


FIGURE 6 – Le graphe P_3

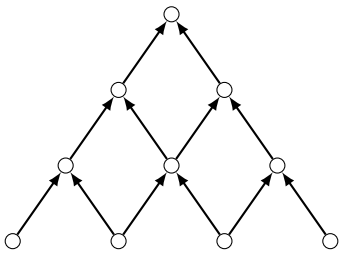


FIGURE 7 – Le graphe P_4

Une représentation des graphes P_3 et P_4 est donnée en figures 6 et 7.

Question 38 Montrer que pour $k > 0$, il existe une stratégie en espace k pour le graphe P_k. Quel est le temps minimal pour une telle stratégie ?

On cherche à montrer dans les deux questions suivantes qu'une telle stratégie est optimale.

On appelle p le puits de P_k. On considère une stratégie en temps T, c'est-à-dire utilisant T mouvements $(m_1, \ldots, m_T)$, et en espace M. On définit l'ensemble suivant :

$$J = \{j \in \{1, 2, \ldots, T\} \mid \text{après } m_j, \text{ tous les chemins d'une source de } P_k \text{ à } p \text{ contiennent un sommet marqué}\}$$

Question 39 Montrer que J est non vide. On note j_0 son minimum. Montrer que le mouvement m_{j_0} a marqué une source s_0 de P_k.

Question 40 Montrer que M est supérieur ou égal à la longueur du chemin de s_0 à p. En déduire que $M_{\min}(P_k) = k$ pour $k > 0$.

3.3 Une borne supérieure en espace

L'étude des graphes pyramides a permis de montrer que pour $n > 0$, il existe des graphes à n sommets qui ne possèdent pas de stratégie en espace inférieur à $\sqrt{2n}$. Il s'agit d'une borne inférieure de l'espace minimal requis pour une stratégie de certains graphes à n sommets. En 1976, W. Paul, R. Tarjan et J. Celoni ont amélioré cette borne inférieure en construisant, pour tout $n > 0$, un graphe à n sommets dont toute stratégie nécessite un espace de l'ordre de $\dfrac{n}{\log n}$.

Dans cette partie, on cherche à montrer que sous certaines hypothèses de degrés entrants, cette borne est optimale. On considère pour la suite uniquement des graphes orientés sans cycles dont les degrés entrants de tous les sommets sont inférieurs ou égaux à 2, en d'autres termes, tels que chaque sommet a zéro, un ou deux prédécesseurs.

On veut montrer que tout graphe à n sommets possède une stratégie en espace $\mathcal{O}\left(\dfrac{n}{\log n}\right)$.

On note, pour $m > 0$, $a(m)$ le nombre d'arêtes minimal d'un graphe G tel que $M_{\min}(G) = m$. Dans un souci de simplification de la preuve, jusqu'à la question 48 incluse, on supposera que m est un multiple de 4.

Question 41 Montrer que a est une fonction croissante.

On considère un graphe $G = (S, A)$ tel que $M_{\min}(G) = m$, on note T_1 l'ensemble des sommets qui peuvent être marqués en utilisant $\dfrac{m}{2}$ marqueurs ou moins, et $T_2 = S \setminus T_1$. Par définition, tous les sommets de T_2 nécessitent strictement plus que $\dfrac{m}{2}$ marqueurs. On note de plus $B_1 = A \cap (T_1 \times T_1)$, $B_2 = A \cap (T_2 \times T_2)$, et $B = A \setminus (B_1 \cup B_2)$. Enfin, on note $H_1 = (T_1, B_1)$ et $H_2 = (T_2, B_2)$.

Question 42 Montrer que pour toute arête $(s, t) \in B$, $s \in T_1$ et $t \in T_2$.

Question 43 Montrer que pour toute stratégie pour un jeu restreint au graphe H_1, il existe un sommet de T_1 qui nécessite $\dfrac{m}{2} - 1$ marqueurs ou plus pour être marqué.

On pourra raisonner par l'absurde en considérant un sommet $s \in T_2$ dont tous les prédécesseurs dans le graphe G sont dans T_1.

Question 44 On cherche à montrer que pour une stratégie pour un jeu restreint au graphe H_2, il existe un sommet qui nécessite $\frac{m}{2} - 1$ marqueurs ou plus. Pour ce faire, on raisonne par l'absurde, et on suppose pour cette question que tous les sommets peuvent être marqués dans H_2 en utilisant $\frac{m}{2} - 2$ marqueurs ou moins. Montrer que dans un jeu dans G, tous les sommets de S peuvent être marqués en utilisant $m - 1$ marqueurs ou moins et conclure.

On suppose pour les trois questions suivantes que $|B| < \frac{m}{4}$.

Question 45 Montrer que dans un jeu dans le graphe H_1, tous les prédécesseurs dans T_1 d'un sommet de T_2 peuvent être marqués simultanément en utilisant $\frac{m}{2} + \frac{m}{4} - 1 \leqslant \frac{3m}{4}$ marqueurs ou moins.

Question 46 En déduire qu'une stratégie dans le graphe H_2 nécessite $\frac{3m}{4}$ marqueurs ou plus.

Question 47 Montrer qu'il existe un ensemble $C \subset B_2$, de cardinal $\frac{m}{4}$ ou plus, tel qu'une stratégie sur le graphe $(T_2, B_2 \setminus C)$ nécessite $\frac{m}{2}$ marqueurs ou plus.

On revient dans le cas général sans hypothèse sur $|B|$.

Question 48 Montrer que dans tous les cas, $a(m) \geqslant 2a\left(\frac{m}{2} - 1\right) + \frac{m}{4}$.

On suppose que le résultat de la question 48 est vrai même lorsque m n'est pas un multiple de 4.

Question 49 Montrer que pour tout $m > 0$, $a(m) \geqslant \frac{m}{8} \log_2 m$. En déduire qu'un graphe qui nécessite m marqueurs ou plus pour être marqué possède $\frac{m}{16} \log_2 m$ sommets ou plus. Conclure que, pour n assez grand, tout graphe à n sommets possède une stratégie en espace au plus $\frac{32n}{\log_2 n}$.